INVENTOR DECLARATION UNDER 37C.F.R. §131

I, Dan MingLun Chuang, state that:

I am one of the inventors of application serial number 10/673,678, filed on September 29, 2003;

I am the inventor of Provisional Application 60/415,320, filed September 30, 2002 and incorporated by reference in the 10/673,678 application;

The Provisional Application included the following documents (copies attached):

"PSN Node Programmable Scalar Node," and

"Adaptive Integrated Circuitry with Heterogenous and Reconfigurable Matrices of Diverse and Adaptive Computational Units Having Fixed, Application Specifics Computational Elements;"

I made the invention disclosed in the Provisional Application in the US prior to September 26, 2002 including;

I conceived the invention prior to September 26, 2002; and

I diligently caused the above Provisional Application to be filed in the USPTO.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and, further, that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Sec. 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patents issued thereon.

7/12/07
Date

Dan MingLun Chuang

ADAPTIVE INTEGRATED CIRCUITRY WITH HETEROGENEOUS AND RECONFIGURABLE MATRICES OF DIVERSE AND ADAPTIVE COMPUTATIONAL UNITS HAVING FIXED, APPLICATION SPECIFIC COMPUTATIONAL ELEMENTS

Field of the Invention

The present invention relates, in general, to integrated circuits and, more particularly, to adaptive integrated circuitry with heterogeneous and reconfigurable matrices of diverse and adaptive computational units having fixed, application specific computational elements.

Background of the Invention

The advances made in the design and development of integrated circuits ("ICs") have generally produced ICs of several different types or categories having different properties and functions, such as the class of universal Turing machines (including microprocessors and digital signal processors ("DSPs")), application specific integrated circuits ("ASICs"), and field programmable gate arrays ("FPGAs"). Each of these different types of ICs, and their corresponding design methodologies, have distinct advantages and disadvantages.

Microprocessors and DSPs, for example, typically provide a flexible, software programmable solution for the implementation of a wide variety of tasks. As various technology standards evolve, microprocessors and DSPs may be reprogrammed, to varying degrees, to perform various new or altered functions or operations. Various tasks or algorithms, however, must be partitioned and constrained to fit the physical limitations of the processor, such as bus widths and hardware availability. In addition, as processors are designed for the execution of instructions, large areas of the IC are allocated to instruction processing, with the result that the processors are comparatively inefficient in the performance of actual algorithmic operations, with only a few percent of these operations performed during any given clock cycle. Microprocessors and DSPs, moreover, have a comparatively limited activity factor, such as having only approximately five percent of their transistors engaged in algorithmic operations at any given time, with most of the transistors allocated to instruction processing. As a consequence, for the performance of any given algorithmic operation, processors consume significantly more IC (or silicon) area and consume significantly more power compared to other types of ICs, such as ASICs.

While having comparative advantages in power consumption and size, ASICs provide a fixed, rigid or "hard-wired" implementation of transistors (or logic gates) for the performance of a highly specific task or a group of highly specific tasks. ASICs typically perform these tasks quite effectively, with a comparatively high activity factor, such as with twenty-five to thirty percent of the transistors engaged in switching at any given time. Once etched, however, an ASIC is not readily changeable, with any modification being time-consuming and expensive, effectively requiring new masks and new fabrication. As a further result, ASIC design virtually always has a degree of obsolescence, with a design cycle lagging behind the evolving standards for product implementations. For example, an ASIC designed to implement GSM or CDMA standards for mobile communication becomes relatively obsolete with the advent of a new standard, such as 3G.

FPGAs have evolved to provide some design and programming flexibility, allowing a degree of post-fabrication modification. FPGAs typically consist of small, identical sections or "islands" of programmable logic (logic gates) surrounded by many levels of programmable interconnect, and may include memory elements. FPGAs are homogeneous, with the IC comprised of repeating arrays of identical groups of logic gates, memory and programmable interconnect. A particular function may be implemented by configuring (or reconfiguring) the interconnect to connect the various logic gates in particular sequences and arrangements. The most significant advantage of FPGAs are their post-fabrication reconfigurability, allowing a degree of flexibility in the implementation of changing or evolving specifications or standards. The reconfiguring process for an FPGA is comparatively slow, however, and is typically unsuitable for most real-time, immediate applications.

While this post-fabrication flexibility of FPGAs provides a significant advantage, FPGAs have corresponding and inherent disadvantages. Compared to ASICs, FPGAs are very expensive and very inefficient for implementation of particular functions, and are often subject to a "combinatorial explosion" problem. More particularly, for FPGA implementation, an algorithmic operation comparatively may require orders of magnitude more IC area, time and power, particularly when the particular algorithmic operation is a poor fit to the pre-existing, homogeneous islands of logic gates of the FPGA material. In addition, the programmable interconnect, which should be sufficiently rich and available to provide reconfiguration flexibility, has a correspondingly high capacitance, resulting in comparatively slow operation and high power consumption. For example, compared to an ASIC, an FPGA implementation of a relatively simple function, such as a multiplier, consumes significant IC area and vast amounts of power, while providing significantly poorer performance by several orders of magnitude. In addition, there is a chaotic element to FPGA routing, rendering FPGAs subject to unpredictable routing delays and wasted logic resources, typically with approximately one-half or more of the theoretically available gates remaining unusable due to limitations in routing resources and routing algorithms.

Various prior art attempts to meld or combine these various processor, ASIC and FPGA architectures have had utility for certain limited applications, but have not proven to be successful or useful for low power, high efficiency, and real-time applications. Typically, these prior art attempts have simply provided, on a single chip, an area of known FPGA material (consisting of a repeating array of identical logic gates with interconnect) adjacent to either a processor or an ASIC, with limited interoperability, as an aid to either processor or ASIC functionality. For example, Trimberger U. S. Patent No. 5,737,631, entitled "Reprogrammable Instruction Set Accelerator", issued April 7, 1998, is designed to provide instruction acceleration for a general purpose processor, and merely discloses a host CPU made up of such a basic microprocessor combined in parallel with known FPGA material (with an FPGA configuration store, which together form the reprogrammable instruction set accelerator). This reprogrammable instruction set accelerator, while allowing for some post-fabrication reconfiguration flexibility and processor acceleration, is nonetheless subject to the various disadvantages of traditional processors and traditional FPGA material, such as high power consumption and high capacitance, with comparatively low speed, low efficiency and low activity factors.

Tavana et al. U. S. Patent No. 6,094,065, entitled "Integrated Circuit with Field Programmable and Application Specific Logic Areas", issued July 25, 2000, is designed to allow a degree of post-fabrication modification of an ASIC, such as for correction of design or other layout flaws, and discloses use of a field programmable gate array in a parallel combination with a mask-defined application specific logic area (*i.e.,* ASIC material). Once again, known FPGA material, consisting of a repeating array of identical logic gates within a rich programmable interconnect, is merely placed adjacent to ASIC material within the same silicon chip. While potentially providing post-fabrication means for "bug fixes" and other error correction, the prior art IC is nonetheless subject to the various disadvantages of traditional ASICs and traditional FPGA material, such as highly limited reprogrammability of an ASIC, combined with high power consumption, comparatively low speed, low efficiency and low activity factors of FPGAs.

As a consequence, a need remains for a new form or type of integrated circuitry which effectively and efficiently combines and maximizes the various advantages of processors, ASICs and FPGAs, while minimizing potential disadvantages. Such a new form or type of integrated circuit should include, for instance, the programming flexibility of a processor, the post-fabrication flexibility of FPGAs, and the high speed and high utilization factors of an ASIC. Such integrated circuitry should be readily reconfigurable, in real-time, and be capable of having corresponding, multiple modes of operation. In addition, such integrated circuitry should minimize power consumption and should be suitable for low power applications, such as for use in hand-held and other battery-powered devices.

Summary of the Invention

The present invention provides new form or type of integrated circuitry which effectively and efficiently combines and maximizes the various advantages of processors, ASICs and FPGAs, while minimizing potential disadvantages. In accordance with the present invention, such a new form or type of integrated circuit, referred to as an adaptive computing engine (ACE), is disclosed which provides the programming flexibility of a processor, the post-fabrication flexibility of FPGAs, and the high speed and high utilization factors of an ASIC. The ACE integrated circuitry of the present invention is readily reconfigurable, in real-time, is capable of having corresponding, multiple modes of operation, and further minimizes power consumption while increasing performance, with particular suitability for low power applications, such as for use in hand-held and other battery-powered devices.

The ACE architecture of the present invention, for adaptive or reconfigurable computing, includes a plurality of heterogeneous computational elements coupled to an interconnection network, rather than the homogeneous units of FPGAs. The plurality of heterogeneous computational elements include corresponding computational elements having fixed and differing architectures, such as fixed architectures for different functions such as memory, addition, multiplication, complex multiplication, subtraction, configuration, reconfiguration, control, input, output, and field programmability. In response to configuration information, the interconnection network is operative in real-time to configure and reconfigure the plurality of heterogeneous computational elements for a plurality of different functional modes, including linear algorithmic operations, non-linear algorithmic operations, finite state machine operations, memory operations, and bit-level manipulations.

As illustrated and discussed in greater detail below, the ACE architecture of the present invention provides a single IC, which may be configured and reconfigured in real-time, using these fixed and application specific computation elements, to perform a wide variety of tasks. For example, utilizing differing configurations over time of the same set of heterogeneous computational elements, the ACE architecture may implement functions such as finite impulse response filtering, fast Fourier transformation, discrete cosine transformation, and with other types of computational elements, may implement many other high level processing functions for advanced communications and computing.

Numerous other advantages and features of the present invention will become readily apparent from the following detailed description of the invention and the embodiments thereof, from the claims and from the accompanying drawings.

Brief Description of the Drawings

Figure 1 is a block diagram illustrating a preferred apparatus embodiment in accordance with the present invention.

Figure 2 is a schematic diagram illustrating an exemplary data flow graph in accordance with the present invention.

Figure 3 is a block diagram illustrating a reconfigurable matrix, a plurality of computation units, and a plurality of computational elements, in accordance with the present invention.

Figure 4 is a block diagram illustrating, in greater detail, a computational unit of a reconfigurable matrix in accordance with the present invention.

Figures 5A through 5E are block diagrams illustrating, in detail, exemplary fixed and specific computational elements, forming computational units, in accordance with the present invention.

Figure 6 is a block diagram illustrating, in detail, a preferred multi-function adaptive computational unit having a plurality of different, fixed computational elements, in accordance with the present invention.

Figure 7 is a block diagram illustrating, in detail, a preferred adaptive logic processor computational unit having a plurality of fixed computational elements, in accordance with the present invention.

Figure 8 is a block diagram illustrating, in greater detail, a preferred core cell of an adaptive logic processor computational unit with a fixed computational element, in accordance with the present invention.

Figure 9 is a block diagram illustrating, in greater detail, a preferred fixed computational element of a core cell of an adaptive logic processor computational unit, in accordance with the present invention.

Detailed Description of the Invention

While the present invention is susceptible of embodiment in many different forms, there are shown in the drawings and will be described herein in detail specific embodiments thereof, with the understanding that the present disclosure is to be considered as an exemplification of the principles of the invention and is not intended to limit the invention to the specific embodiments illustrated.

As indicated above, a need remains for a new form or type of integrated circuitry which effectively and efficiently combines and maximizes the various advantages of processors, ASICs and FPGAs, while minimizing potential disadvantages. In accordance with the present invention, such a new form or type of integrated circuit, referred to as an adaptive computing engine (ACE), is disclosed which provides the programming flexibility of a processor, the post-fabrication flexibility of FPGAs, and the high speed and high utilization factors of an ASIC. The ACE integrated circuitry of the present invention is readily reconfigurable, in real-time, is capable of having corresponding, multiple modes of operation, and further minimizes power consumption while increasing performance, with particular suitability for low power applications.

Figure 1 is a block diagram illustrating a preferred apparatus 100 embodiment in accordance with the present invention. The apparatus 100, referred to herein as an adaptive computing engine ("ACE") 100, is preferably embodied as an integrated circuit, or as a portion of an integrated circuit having other, additional components. In the preferred embodiment, and as discussed in greater detail below, the ACE 100 includes one or more reconfigurable matrices (or nodes)150, such as matrices 150A through 150N as illustrated, and a matrix interconnection network 110. Also in the preferred embodiment, and as discussed in detail below, one or more of the matrices 150, such as matrices 150A and 150B, are configured for functionality as a controller 120, while other matrices, such as matrices 150C and 150D, are configured for functionality as a memory 140. The various matrices 150 and matrix interconnection network 110 may also be implemented together as fractal subunits, which may be scaled from a few nodes to thousands of nodes.

A significant departure from the prior art, the ACE 100 does not utilize traditional (and typically separate) data, DMA, random access, configuration and instruction busses for signaling and other transmission between and among the reconfigurable matrices 150, the controller 120, and the memory 140, or for other input/output ("I/O") functionality. Rather, data, control and configuration information are transmitted between and among these matrix 150 elements, utilizing the matrix interconnection network 110, which may be configured and reconfigured, in real-time, to provide any given connection between and among the reconfigurable matrices 150, including those matrices 150 configured as the controller 120 and the memory 140, as discussed in greater detail below.

The matrices 150 configured to function as memory 140 may be implemented in any desired or preferred way, utilizing computational elements (discussed below) of fixed memory elements, and may be included within the ACE 100 or incorporated within another IC or portion of an IC. In the preferred embodiment, the memory 140 is included within the ACE 100, and preferably is comprised of computational elements which are low power consumption random access memory (RAM), but also may be comprised of computational elements of any other form of memory, such as flash, DRAM, SRAM, MRAM, ROM, EPROM or E^2PROM. In the preferred embodiment, the memory 140 preferably includes direct memory access (DMA) engines, not separately illustrated.

The controller 120 is preferably implemented, using matrices 150A and 150B configured as adaptive finite state machines, as a reduced instruction set ("RISC") processor, controller or other device or IC capable of performing the two types of functionality discussed below. (Alternatively, these functions may be implemented utilizing a conventional RISC or other processor.) The first control functionality, referred to as "kernal" control, is illustrated as kernal controller ("KARC") of matrix 150A, and the second control functionality, referred to as "matrix" control, is illustrated as matrix controller ("MARC") of matrix 150B. The kernal and matrix control functions of the controller 120 are explained in greater detail below, with reference to the configurability and reconfigurability of the various matrices 150, and with reference to the preferred form of combined data, configuration and control information referred to herein as a "silverware" module.

The matrix interconnection network 110 of Figure 1, and its subset interconnection networks separately illustrated in Figures 3 and 4 (Boolean interconnection network 210, data interconnection network 240, and interconnect 220), collectively and generally referred to herein as "interconnect", "interconnection(s)" or "interconnection network(s)", may be implemented generally as known in the art, such as utilizing FPGA interconnection networks or switching fabrics, albeit in a considerably more varied fashion. In the preferred embodiment, the various interconnection networks are implemented as described, for example, in U.S. Patent No. 5,218,240, U.S. Patent No. 5,336,950, U.S. Patent No. 5,245,227, and U.S. Patent No. 5,144,166, and also as discussed below and as illustrated with reference to Figures 7, 8 and 9. These various interconnection networks provide selectable (or switchable) connections between and among the controller 120, the memory 140, the various matrices 150, and the computational units 200 and computational elements 250 discussed below, providing the physical basis for the configuration and reconfiguration referred to herein, in response to and under the control of configuration signaling generally referred to herein as "configuration information". In addition, the various interconnection networks (110, 210, 240 and 220) provide selectable or switchable data, input, output, control and configuration paths, between and among the controller 120, the memory 140, the various matrices 150, and the computational units 200 and computational elements 250, in lieu of any form of traditional or separate input/output busses, data busses, DMA, RAM, configuration and instruction busses.

It should be pointed out, however, that while any given switching or selecting operation of or within the various interconnection networks (110, 210, 240 and 220) may be implemented as known in the art, the design and layout of the various interconnection networks (110, 210, 240 and 220), in accordance with the present invention, are new and novel, as discussed in greater detail below. For example, varying levels of interconnection are provided to correspond to the varying levels of the matrices 150, the computational units 200, and the computational elements 250, discussed below. At the matrix 150 level, in comparison with the prior art FPGA interconnect, the matrix interconnection network 110 is considerably more limited and less "rich", with lesser connection capability in a given area, to reduce capacitance and increase speed of operation. Within a particular matrix 150 or computational unit 200, however, the interconnection network (210, 220 and 240) may be considerably more dense and rich, to provide greater adaptation and reconfiguration capability within a narrow or close locality of reference.

The various matrices or nodes 150 are reconfigurable and heterogeneous, namely, in general, and depending upon the desired configuration: reconfigurable matrix 150A is generally different from reconfigurable matrices 150B through 150N; reconfigurable matrix 150B is generally different from reconfigurable matrices 150A and 150C through 150N; reconfigurable matrix 150C is generally different from reconfigurable matrices 150A, 150B and 150D through 150N, and so on. The various reconfigurable matrices 150 each generally contain a different or varied mix of adaptive and reconfigurable computational (or computation) units (200); the computational units 200, in turn, generally contain a different or varied mix of fixed, application specific computational elements (250), discussed in greater detail below with reference to Figures 3 and 4, which may be adaptively connected, configured and reconfigured in various ways to perform varied functions, through the various interconnection networks. In addition to varied internal configurations and reconfigurations, the various matrices 150 may be connected, configured and reconfigured at a higher level, with respect to each of the other matrices 150, through the matrix interconnection network 110, also as discussed in greater detail below.

Several different, insightful and novel concepts are incorporated within the ACE 100 architecture of the present invention, and provide a useful explanatory basis for the real-time operation of the ACE 100 and its inherent advantages.

The first novel concepts of the present invention concern the adaptive and reconfigurable use of application specific, dedicated or fixed hardware units (computational elements 250), and the selection of particular functions for acceleration, to be included within these application specific, dedicated or fixed hardware units (computational elements 250) within the computational units 200 (Fig. 3) of the matrices 150, such as pluralities of multipliers, complex multipliers, and adders, each of which are designed for optimal execution of corresponding multiplication, complex multiplication, and addition functions. Given that the ACE 100 is to be optimized, in the preferred embodiment, for low power consumption, the functions for acceleration are selected based upon power consumption. For example, for a given application such as mobile communication, corresponding C (C+ or C++) or other code may be analyzed for power consumption. Such empirical analysis may reveal, for example, that a small portion of such code, such as 10%, actually consumes 90% of the operating power when executed. In accordance with the present invention, on the basis of such power utilization, this small portion of code is selected for acceleration within certain types of the reconfigurable matrices 150, with the remaining code, for example, adapted to run within matrices 150 configured as controller 120. Additional code may also be selected for acceleration, resulting in an optimization of power consumption by the ACE 100, up to any potential trade-off resulting from design or operational complexity. In addition, as discussed with respect to Figure 3, other functionality, such as control code, may be accelerated within matrices 150 when configured as finite state machines.

Next, algorithms or other functions selected for acceleration are converted into a form referred to as a "data flow graph" ("DFG"). A schematic diagram of an exemplary data flow graph, in accordance with the present invention, is illustrated in Figure 2. As illustrated in Fig. 2, an algorithm or function useful for CDMA voice coding (QCELP (Qualcomm code excited linear prediction) is implemented utilizing four multipliers 190 followed by four adders 195. Through the varying levels of interconnect, the algorithms of this data flow graph are then implemented, at any given time, through the configuration and reconfiguration of fixed computational elements (250), namely, implemented within hardware which has been optimized and configured for efficiency, *i.e.,* a "machine" is configured in real-time which is optimized to perform the particular algorithm. Continuing with the exemplary DFG or Figure 2, four fixed or dedicated multipliers, as computational elements 250, and four fixed or dedicated adders, also as different computational elements 250, are configured in real-time through the interconnect to perform the functions or algorithms of the particular DFG.

The third and perhaps most significant concept of the present invention, and a marked departure from the concepts and precepts of the prior art, is the concept of reconfigurable "heterogeneity" utilized to implement the various selected algorithms mentioned above. As indicated above, prior art reconfigurability has relied exclusively on homogeneous FPGAs, in which identical blocks of logic gates are repeated as an array within a rich, programmable interconnect, with the interconnect subsequently configured to provide connections between and among the identical gates to implement a particular function, albeit inefficiently and often with routing and combinatorial problems. In stark contrast, in accordance with the present invention, within computation units 200, different computational elements (250) are implemented directly as correspondingly different fixed (or dedicated) application specific hardware, such as dedicated multipliers, complex multipliers, and adders. Utilizing interconnect (210 and 220), these differing, heterogeneous computational elements (250) may then be adaptively configured, in real-time, to perform the selected algorithm, such as the performance of discrete cosine transformations often utilized in mobile communications. For the data flow graph example of Fig. 2, four multipliers and four adders will be configured, *i.e.,* connected in real-time, to perform the particular algorithm. As a consequence, in accordance with the present invention, different ("heterogeneous") computational elements (250) are configured and reconfigured, at any given time, to optimally perform a given algorithm or other function. In addition, for repetitive functions, a given instantiation or configuration of computational elements may also remain in place over time, *i.e.,* unchanged, throughout the course of such repetitive calculations.

The temporal nature of the ACE 100 architecture should also be noted. At any given instant of time, utilizing different levels of interconnect (110, 210, 240 and 220), a particular configuration may exist within the ACE 100 which has been optimized to perform a given function or implement a particular algorithm. At another instant in time, the configuration may be changed, to interconnect other computational elements (250) or connect the same computational elements 250 differently, for the performance of another function or algorithm. Two important features arise from this temporal reconfigurability. First, as algorithms may change over time to, for example, implement a new technology standard, the ACE 100 may co-evolve and be reconfigured to implement the new algorithm. For a simplified example, a fifth multiplier and a fifth adder may be incorporated into the DFG of Fig. 2 to execute a correspondingly new algorithm, with additional interconnect also potentially utilized to implement any additional bussing functionality. Second, because computational elements are interconnected at one instant in time, as an instantiation of a given algorithm, and then reconfigured at another instant in time for performance of another, different algorithm, gate (or transistor) utilization is maximized, providing significantly better performance than the most efficient ASICs relative to their activity factors.

This temporal reconfigurability of computational elements 250, for the performance of various different algorithms, also illustrates a conceptual distinction utilized herein between configuration and reconfiguration, on the one hand, and programming or reprogrammability, on the other hand. Typical programmability utilizes a pre-existing group or set of functions, which may be called in various orders, over time, to implement a particular algorithm. In contrast, configurability and reconfigurability, as used herein, includes the additional capability of adding or creating new functions which were previously unavailable or non-existent.

Next, the present invention also utilizes a tight coupling (or interdigitation) of data and configuration (or other control) information, within one, effectively continuous stream of information. This coupling or commingling of data and configuration information, referred to as a "silverware" module, is the subject of a separate, related patent application. For purposes of the present invention, however, it is sufficient to note that this coupling of data and configuration information into one information (or bit) stream helps to enable real-time reconfigurability of the ACE 100, without a need for the (often unused) multiple, overlaying networks of hardware interconnections of the prior art. For example, as an analogy, a particular, first configuration of computational elements at a particular, first period of time, as the hardware to execute a corresponding algorithm during or after that first period of time, may be viewed or conceptualized as a hardware analog of "calling" a subroutine in software which may perform the same algorithm. As a consequence, once the configuration of the computational elements has occurred (*i.e.,* is in place), as directed by the configuration information, the data for use in the algorithm is immediately available as part of the silverware module. The same computational elements may then be reconfigured for a second period of time, as directed by second configuration information, for execution of a second, different algorithm, also utilizing immediately available data. The immediacy of the data, for use in the configured computational elements, provides a one or two clock cycle hardware analog to the multiple and separate software steps of determining a memory address and fetching stored data from the addressed registers. This has the further result of additional efficiency, as the configured computational elements may execute, in comparatively few clock cycles, an algorithm which may require orders of magnitude more clock cycles for execution if called as a subroutine in a conventional microprocessor or DSP.

This use of silverware modules, as a commingling of data and configuration information, in conjunction with the real-time reconfigurability of a plurality of heterogeneous and fixed computational elements 250 to form adaptive, different and heterogenous computation units 200 and matrices 150, enables the ACE 100 architecture to have multiple and different modes of operation. For example, when included within a hand-held device, given a corresponding silverware module, the ACE 100 may have various and different operating modes as a cellular or other mobile telephone, a music player, a pager, a personal digital assistant, and other new or existing functionalities. In addition, these operating modes may change based upon the physical location of the device; for example, when configured as a CDMA mobile telephone for use in the United States, the ACE 100 may be reconfigured as a GSM mobile telephone for use in Europe.

Referring again to Figure 1, the functions of the controller 120 (preferably matrix (KARC) 150A and matrix (MARC) 150B, configured as finite state machines) may be explained with reference to a silverware module, namely, the tight coupling of data and configuration information within a single stream of information, with reference to multiple potential modes of operation, with reference to the reconfigurable matrices 150, and with reference to the reconfigurable computation units 200 and the computational elements 150 illustrated in Fig. 3. As indicated above, through a silverware module, the ACE 100 may be configured or reconfigured to perform a new or additional function, such as an upgrade to a new technology standard or the addition of an entirely new function, such as the addition of a music function to a mobile communication device. Such a silverware module may be stored in the matrices 150 of memory 140, or may be input from an external (wired or wireless) source through, for example, matrix interconnection network 110. In the preferred embodiment, one of the plurality of matrices 150 is configured to decrypt such a module and verify its validity, for security purposes. Next, prior to any configuration or reconfiguration of existing ACE 100 resources, the controller 120, through the matrix (KARC) 150A, checks and verifies that the configuration or reconfiguration may occur without adversely affecting any pre-existing functionality, such as whether the addition of music functionality would adversely affect pre-existing mobile communications functionality. In the preferred embodiment, the system requirements for such configuration or reconfiguration are included within the silverware module, for use by the matrix (KARC) 150A in performing this evaluative function. If the configuration or reconfiguration may occur without such adverse affects, the silverware module is allowed to load into the matrices 150 of memory 140, with the matrix (KARC) 150A setting up the DMA engines within the matrices 150C and 150D of the memory 140 (or other stand-alone DMA engines of a conventional memory). If the configuration or reconfiguration would or may have such adverse affects, the matrix (KARC) 150A does not allow the new module to be incorporated within the ACE 100.

Continuing to refer to Figure 1, the matrix (MARC) 150B manages the scheduling of matrix 150 resources and the timing of any corresponding data, to synchronize any configuration or reconfiguration of the various computational elements 250 and computation units 200 with any corresponding input data and output data. In the preferred embodiment, timing information is also included within a silverware module, to allow the matrix (MARC) 150B through the various interconnection networks to direct a reconfiguration of the various matrices 150 in time, and preferably just in time, for the reconfiguration to occur before corresponding data has appeared at any inputs of the various reconfigured computation units 200. In addition, the matrix (MARC) 150B may also perform any residual processing which has not been accelerated within any of the various matrices 150. As a consequence, the matrix (MARC) 150B may be viewed as a control unit which "calls" the configurations and reconfigurations of the matrices 150, computation units 200 and computational elements 250, in real-time, in synchronization with any corresponding data to be utilized by these various reconfigurable hardware units, and which performs any residual or other control processing. Other matrices 150 may also include this control functionality, with any given matrix 150 capable of calling and controlling a configuration and reconfiguration of other matrices 150.

Figure 3 is a block diagram illustrating, in greater detail, a reconfigurable matrix 150 with a plurality of computation units 200 (illustrated as computation units 200A through 200N), and a plurality of computational elements 250 (illustrated as computational elements 250A through 250Z), and provides additional illustration of the preferred types of computational elements 250 and a useful summary of the present invention. As illustrated in Figure 3, any matrix 150 generally includes a matrix controller 230, a plurality of computation (or computational) units 200, and as logical or conceptual subsets or portions of the matrix interconnect network 110, a data interconnect network 240 and a Boolean interconnect network 210. As mentioned above, in the preferred embodiment, at increasing "depths" within the ACE 100 architecture, the interconnect networks become increasingly rich, for greater levels of adaptability and reconfiguration. The Boolean interconnect network 210, also as mentioned above, provides the reconfiguration and data interconnection capability between and among the various computation units 200, and is preferably small (*i.e.,* only a few bits wide), while the data interconnect network 240 provides the reconfiguration and data interconnection capability for data input and output between and among the various computation units 200, and is preferably comparatively large (*i.e.,* many bits wide). It should be noted, however, that while conceptually divided into reconfiguration and data capabilities, any given physical portion of the matrix interconnection network 110, at any given time, may be operating as either the Boolean interconnect network 210, the data interconnect network 240, the lowest level interconnect 220 (between and among the various computational elements 250), or other input, output, or connection functionality.

Continuing to refer to Figure 3, included within a computation unit 200 are a plurality of computational elements 250, illustrated as computational elements 250A through 250Z (individually and collectively referred to as computational elements 250), and additional interconnect 220. The interconnect 220 provides the reconfigurable interconnection capability and input/output paths between and among the various computational elements 250. As indicated above, each of the various computational elements 250 consist of dedicated, application specific hardware designed to perform a given task or range of tasks, resulting in a plurality of different, fixed computational elements 250. Utilizing the interconnect 220, the fixed computational elements 250 may be reconfigurably connected together into adaptive and varied computational units 200, which also may be further reconfigured and interconnected, to execute an algorithm or other function, at any given time, such as the quadruple multiplications and additions of the DFG of Fig. 2, utilizing the interconnect 220, the Boolean network 210, and the matrix interconnection network 110.

In the preferred embodiment, the various computational elements 250 are designed and grouped together, into the various adaptive and reconfigurable computation units 200 (as illustrated, for example, in Figures 5A through 9). In addition to computational elements 250 which are designed to execute a particular algorithm or function, such as multiplication or addition, other types of computational elements 250 are also utilized in the preferred embodiment. As illustrated in Fig. 3, computational elements 250A and 250B implement memory, to provide local memory elements for any given calculation or processing function (compared to the more "remote" memory 140). In addition, computational elements 250I, 250J, 250K and 250L are configured to implement finite state machines (using, for example, the computational elements illustrated in Figures 7, 8 and 9), to provide local processing capability (compared to the more "remote" matrix (MARC) 150B), especially suitable for complicated control processing.

With the various types of different computational elements 250 which may be available, depending upon the desired functionality of the ACE 100, the computation units 200 may be loosely categorized. A first category of computation units 200 includes computational elements 250 performing linear operations, such as multiplication, addition, finite impulse response filtering, and so on (as illustrated below, for example, with reference to Figures 5A through 5E and Figure 6). A second category of computation units 200 includes computational elements 250 performing non-linear operations, such as discrete cosine transformation, trigonometric calculations, and complex multiplications. A third type of computation unit 200 implements a finite state machine, such as computation unit 200C as illustrated in Figure 3 and as illustrated in greater detail below with respect to Figures 7 through 9), particularly useful for complicated control sequences, dynamic scheduling, and input/output management, while a fourth type may implement memory and memory management, such as computation unit 200A as illustrated in Fig. 3. Lastly, a fifth type of computation unit 200 may be included to perform bit-level manipulation, such as for encryption, decryption, channel coding, Viterbi decoding, and packet and protocol processing (such as Internet Protocol processing).

In the preferred embodiment, in addition to control from other matrices or nodes 150, a matrix controller 230 may also be included within any given matrix 150, also to provide greater locality of reference and control of any reconfiguration processes and any corresponding data manipulations. For example, once a reconfiguration of computational elements 250 has occurred within any given computation unit 200, the matrix controller 230 may direct that that particular instantiation (or configuration) remain intact for a certain period of time to, for example, continue repetitive data processing for a given application.

Figure 4 is a block diagram illustrating, in greater detail, an exemplary or representative computation unit 200 of a reconfigurable matrix 150 in accordance with the present invention. As illustrated in Figure 4, a computation unit 200 typically includes a plurality of diverse, heterogeneous and fixed computational elements 250, such as a plurality of memory computational elements 250A and 250B, and forming a computational unit ("CU") core 260, a plurality of algorithmic or finite state machine computational elements 250C through 250K. As discussed above, each computational element 250, of the plurality of diverse computational elements 250, is a fixed or dedicated, application specific circuit, designed and having a corresponding logic gate layout to perform a specific function or algorithm, such as addition or multiplication. In addition, the various memory computational elements 250A and 250B may be implemented with various bit depths, such as RAM (having significant depth), or as a register, having a depth of 1 or 2 bits.

Forming the conceptual data and Boolean interconnect networks 240 and 210, respectively, the exemplary computation unit 200 also includes a plurality of input multiplexers 280, a plurality of input lines (or wires) 281, and for the output of the CU core 260 (illustrated as line or wire 270), a plurality of output demultiplexers 285 and 290, and a plurality of output lines (or wires) 291. Through the input multiplexers 280, an appropriate input line 281 may be selected for input use in data transformation and in the configuration and interconnection processes, and through the output demultiplexers 285 and 290, an output or multiple outputs may be placed on a selected output line 291, also for use in additional data transformation and in the configuration and interconnection processes.

In the preferred embodiment, the selection of various input and output lines 281 and 291, and the creation of various connections through the interconnect (210, 220 and 240), is under control of control bits 265 from the computational unit controller 255, as discussed below. Based upon these control bits 265, any of the various input enables 251, input selects 252, output selects 253, MUX selects 254, DEMUX enables 256, DEMUX selects 257, and DEMUX output selects 258, may be activated or deactivated.

The exemplary computation unit 200 includes a computation unit controller 255 which provides control, through control bits 265, over what each computational element 250, interconnect (210, 220 and 240), and other elements (above) does with every clock cycle. Not separately illustrated, through the interconnect (210, 220 and 240), the various control bits 265 are distributed, as may be needed, to the various portions of the computation unit 200, such as the various input enables 251, input selects 252, output selects 253, MUX selects 254, DEMUX enables 256, DEMUX selects 257, and DEMUX output selects 258. The CU controller 295 also includes one or more lines 295 for reception of control (or configuration) information and transmission of status information.

As mentioned above, the interconnect may include a conceptual division into a data interconnect network 240 and a Boolean interconnect network 210, of varying bit widths, as mentioned above. In general, the (wider) data interconnection network 240 is utilized for creating configurable and reconfigurable connections, for corresponding routing of data and configuration information. The (narrower) Boolean interconnect network 210, while also utilized for creating configurable and reconfigurable connections, is utilized for control of logic (or Boolean) decisions of the various data flow graphs, generating decision nodes in such DFGs, and may also be used for data routing within such DFGs.

Figures 5A through 5E are block diagrams illustrating, in detail, exemplary fixed and specific computational elements, forming computational units, in accordance with the present invention. As will be apparent from review of these Figures, many of the same fixed computational elements are utilized, with varying configurations, for the performance of different algorithms.

Figure 5A is a block diagram illustrating a four-point asymmetric finite impulse response (FIR) filter computational unit 300. As illustrated, this exemplary computational unit 300 includes a particular, first configuration of a plurality of fixed computational elements, including coefficient memory 305, data memory 310, registers 315, 320 and 325, multiplier 330, adder 335, and accumulator registers 340, 345, 350 and 355, with multiplexers (MUXes) 360 and 365 forming a portion of the interconnection network (210, 220 and 240).

Figure 5B is a block diagram illustrating a two-point symmetric finite impulse response (FIR) filter computational unit 370. As illustrated, this exemplary computational unit 370 includes a second configuration of a plurality of fixed computational elements, including coefficient memory 305, data memory 310, registers 315, 320 and 325, multiplier 330, adder 335, second adder 375, and accumulator registers 340 and 345, also with multiplexers (MUXes) 360 and 365 forming a portion of the interconnection network (210, 220 and 240).

Figure 5C is a block diagram illustrating a subunit for a fast Fourier transform (FFT) computational unit 400. As illustrated, this exemplary computational unit 400 includes a third configuration of a plurality of fixed computational elements, including coefficient memory 305, data memory 310, registers 315, 320, 325 and 385, multiplier 330, adder 335, and adder/subtractor 380, with multiplexers (MUXes) 360, 365, 390, 395 and 405 forming a portion of the interconnection network (210, 220 and 240).

Figure 5D is a block diagram illustrating a complex finite impulse response (FIR) filter computational unit 440. As illustrated, this exemplary computational unit 440 includes a fourth configuration of a plurality of fixed computational elements, including memory 410, registers 315 and 320, multiplier 330, adder/subtractor 380, and real and imaginary accumulator registers 415 and 420, also with multiplexers (MUXes) 360 and 365 forming a portion of the interconnection network (210, 220 and 240).

Figure 5E is a block diagram illustrating a biquad infinite impulse response (IIR) filter computational unit 450, with a corresponding data flow graph 460. As illustrated, this exemplary computational unit 450 includes a fifth configuration of a plurality of fixed computational elements, including coefficient memory 305, input memory 490, registers 470, 475, 480 and 485, multiplier 330, and adder 335, with multiplexers (MUXes) 360, 365, 390 and 395 forming a portion of the interconnection network (210, 220 and 240).

Figure 6 is a block diagram illustrating, in detail, a preferred multi-function adaptive computational unit 500 having a plurality of different, fixed computational elements, in accordance with the present invention. When configured accordingly, the adaptive computation unit 500 performs each of the various functions previously illustrated with reference to Figures 5A though 5E, plus other functions such as discrete cosine transformation. As illustrated, this multi-function adaptive computational unit 500 includes capability for a plurality of configurations of a plurality of fixed computational elements, including input memory 520, data memory 525, registers 530 (illustrated as registers 530A through 530Q), multipliers 540 (illustrated as multipliers 540A through 540D), adder 545, first arithmetic logic unit (ALU) 550 (illustrated as ALU_1s 550A through 550D), second arithmetic logic unit (ALU) 555 (illustrated as ALU_2s 555A through 555D), and pipeline (length 1) register 560, with inputs 505, lines 515, outputs 570, and multiplexers (MUXes or MXes) 510 (illustrates as MUXes and MXes 510A through 510KK) forming an interconnection network (210, 220 and 240). The two different ALUs 550 and 555 are preferably utilized, for example, for parallel addition and subtraction operations, particularly useful for radix 2 operations in discrete cosine transformation.

Figure 7 is a block diagram illustrating, in detail, a preferred adaptive logic processor (ALP) computational unit 600 having a plurality of fixed computational elements, in accordance with the present invention. The ALP 600 is highly adaptable, and is preferably utilized for input/output configuration, finite state machine implementation, general field programmability, and bit manipulation. The fixed computational element of ALP 600 is a portion (650) of each of the plurality of adaptive core cells (CCs) 610 (Figure 8), as separately illustrated in Figure 9. An interconnection network (210, 220 and 240) is formed from various combinations and permutations of the pluralities of vertical inputs (VIs) 615, vertical repeaters (VRs) 620, vertical outputs (VOs) 625, horizontal repeaters (HRs) 630, horizontal terminators (HTs) 635, and horizontal controllers (HCs) 640.

Figure 8 is a block diagram illustrating, in greater detail, a preferred core cell 610 of an adaptive logic processor computational unit 600 with a fixed computational element 650, in accordance with the present invention. The fixed computational element is a 3input – 2 output function generator 550, separately illustrated in Figure 9. The preferred core cell 610 also includes control logic 655, control inputs 665, control outputs 670 (providing output interconnect), output 675, and inputs (with interconnect muxes) 660 (providing input interconnect).

Figure 9 is a block diagram illustrating, in greater detail, a preferred fixed computational element 650 of a core cell 610 of an adaptive logic processor computational unit 600, in accordance with the present invention. The fixed computational element 650 is comprised of a fixed layout of pluralities of exclusive NOR (XNOR) gates 680, NOR gates 685, NAND gates 690, and exclusive OR (XOR) gates 695, with three inputs 720 and two outputs 710. Configuration and interconnection is provided through MUX 705 and interconnect inputs 730.

As may be apparent from the discussion above, this use of a plurality of fixed, heterogeneous computational elements (250), which may be configured and reconfigured to form heterogeneous computation units (200), which further may be configured and reconfigured to form heterogeneous matrices 150, through the varying levels of interconnect (110, 210, 240 and 220), creates an entirely new class or category of integrated circuit, which may be referred to as an adaptive computing architecture. It should be noted that the adaptive computing architecture of the present invention cannot be adequately characterized, from a conceptual or from a nomenclature point of view, within the rubric or categories of FPGAs, ASICs or processors. For example, the non-

FPGA character of the adaptive computing architecture is immediately apparent because the adaptive computing architecture does not comprise either an array of identical logical units, or more simply, a repeating array of any kind. Also for example, the non-ASIC character of the adaptive computing architecture is immediately apparent because the adaptive computing architecture is not application specific, but provides multiple modes of functionality and is reconfigurable in real-time. Continuing with the example, the non-processor character of the adaptive computing architecture is immediately apparent because the adaptive computing architecture becomes configured, to directly operate upon data, rather than focusing upon executing instructions with data manipulation occurring as a byproduct.

Other advantages of the present invention may be further apparent to those of skill in the art. For mobile communications, for example, hardware acceleration for one or two algorithmic elements has typically been confined to infrastructure base stations, handling many (typically 64 or more) channels. Such as acceleration may be cost justified because increased performance and power savings per channel, performed across multiple channels, results in significant performance and power savings. Such multiple channel performance and power savings are not realizable, using prior art hardware acceleration, in a single operative channel mobile terminal (or mobile unit). In contrast, however, through use of the present invention, cost justification is readily available, given increased performance and power savings, because the same IC area may be configured and reconfigured to accelerate multiple algorithmic tasks, effectively generating or bringing into existence a new hardware accelerator for each next algorithmic element.

Yet additional advantages of the present invention may be further apparent to those of skill in the art. The ACE 100 architecture of the present invention effectively and efficiently combines and maximizes the various advantages of processors, ASICs and FPGAs, while minimizing potential disadvantages. The ACE 100 includes the programming flexibility of a processor, the post-fabrication flexibility of FPGAs, and the high speed and high utilization factors of an ASIC. The ACE 100 is readily reconfigurable, in real-time, and is capable of having corresponding, multiple modes of operation. In addition, through the selection of particular functions for reconfigurable acceleration, the ACE 100 minimizes power consumption and is suitable for low power applications, such as for use in hand-held and other battery-powered devices.

From the foregoing, it will be observed that numerous variations and modifications may be effected without departing from the spirit and scope of the novel concept of the invention. It is to be understood that no limitation with respect to the specific methods and apparatus illustrated herein is intended or should be inferred. It is, of course, intended to cover by the appended claims all such modifications as fall within the scope of the claims.

It is claimed:

1. An adaptive computing integrated circuit, comprising:

a plurality of heterogeneous computational elements, the plurality of heterogeneous computational elements including a first computational element and a second computational element, the first computational element having a first fixed architecture and the second computational element having a second fixed architecture, the first fixed architecture being different than the second fixed architecture; and

an interconnection network coupled to the plurality of heterogeneous computational elements, the interconnection network operative to configure the plurality of heterogeneous computational elements for a first functional mode of a plurality of functional modes, in response to first configuration information, and the interconnection network further operative to reconfigure the plurality of heterogeneous computational elements for a second functional mode of the plurality of functional modes, in response to second configuration information, the first functional mode being different than the second functional mode.

2. The adaptive computing integrated circuit of claim 1, wherein the first fixed architecture and the second fixed architecture are selected from a plurality of specific architectures, the plurality of specific architectures including functions for memory, addition, multiplication, complex multiplication, subtraction, configuration, reconfiguration, control, input, output, and field programmability.

3. The adaptive computing integrated circuit of claim 1, wherein the plurality of functional modes includes linear algorithmic operations, non-linear algorithmic operations, finite state machine operations, memory operations, and bit-level manipulations.

4. The adaptive computing integrated circuit of claim 1, wherein the first fixed architecture and the second fixed architecture are selected to comparatively minimize power consumption of the adaptive computing integrated circuit.

5. The adaptive computing integrated circuit of claim 1, wherein the interconnection network reconfigurably routes data and control information between and among the plurality of heterogeneous computational elements.

6. The adaptive computing integrated circuit of claim 1, wherein the first configuration information and the second configuration information are commingled with data to form a singular bit stream.

7. The adaptive computing integrated circuit of claim 1, further comprising:

a controller coupled to the plurality of heterogeneous computational elements and to the interconnection network, the controller operative to direct and schedule the configuration of the plurality of heterogeneous computational elements for the first functional mode and the reconfiguration of the plurality of heterogeneous computational elements for the second functional mode.

8. The adaptive computing integrated circuit of claim 7, wherein the controller is further operative to time and schedule the configuration and reconfiguration of the plurality of heterogeneous computational elements with corresponding data.

9. The adaptive computing integrated circuit of claim 7, wherein the controller is further operative to select the first configuration information and the second configuration information from a singular bit stream containing data commingled with a plurality of configuration information.

10. The adaptive computing integrated circuit of claim 1, further comprising:

a memory coupled to the plurality of heterogeneous computational elements and to the interconnection network, the memory operative to store the first configuration information and the second configuration information.

11. The adaptive computing integrated circuit of claim 1, wherein the plurality of heterogeneous computational elements may be configured and reconfigured, through the interconnection network and in response to a plurality of configuration information, to implement a plurality of logic functions of a data flow graph.

12. The adaptive computing integrated circuit of claim 1, wherein the interconnection network may be further configured to perform a plurality of logic decisions of a data flow graph.

13. The adaptive computing integrated circuit of claim 1, wherein the plurality of heterogeneous computational elements may be configured to form a plurality of adaptive and heterogeneous computational units.

14. The adaptive computing integrated circuit of claim 13, wherein each computation unit of the plurality of heterogeneous computation units further includes:

a computational unit controller coupled to the plurality of heterogeneous computational elements, the computational unit controller responsive to a plurality of configuration information to generate a plurality of control bits;

a plurality of input multiplexers, the plurality of input multiplexers responsive to the plurality of control bits to select an input line from the interconnection network for the reception of input information; and

a plurality of output demultiplexers, the plurality of output demultiplexers responsive to the plurality of control bits to select a plurality of output lines from the interconnection network for the transfer of output information.

15. The adaptive computing integrated circuit of claim 13, wherein the plurality of computation units may be configured to form a plurality of reconfigurable matrices.

16. The adaptive computing integrated circuit of claim 1, wherein the adaptive computing integrated circuit is embodied within a mobile terminal having a plurality of operating modes.

17. The adaptive computing integrated circuit of claim 16, wherein the plurality of operating modes of the mobile terminal includes mobile telecommunication, personal digital assistance, multimedia reception, mobile packet-based communication, and paging.

18. A method for adaptive computing, the comprising:

in response to first configuration information, configuring through an interconnection network a plurality of heterogeneous computational elements for a first functional mode of a plurality of functional modes, the plurality of heterogeneous computational elements including a first computational element and a second computational element, the first computational element having a first fixed architecture and the second computational element having a second fixed architecture, the first fixed architecture being different than the second fixed architecture; and

in response to second configuration information, reconfiguring through the interconnection network the plurality of heterogeneous computational elements for a second functional mode of the plurality of functional modes, the first functional mode being different than the second functional mode.

19. The adaptive computing method of claim 18, wherein the first fixed architecture and the second fixed architecture are selected from a plurality of specific architectures, the plurality of specific architectures including functions for memory, addition, multiplication, complex multiplication, subtraction, configuration, reconfiguration, control, input, output, and field programmability.

20. The adaptive computing method of claim 18, wherein the plurality of functional modes includes linear algorithmic operations, non-linear algorithmic operations, finite state machine operations, memory operations, and bit-level manipulations.

21. The adaptive computing method of claim 18, wherein the first fixed architecture and the second fixed architecture are selected to comparatively minimize power consumption of the adaptive computing integrated circuit.

22. The adaptive computing method of claim 18, further comprising:

reconfigurably routing, through the interconnection network, data and control information between and among the plurality of heterogeneous computational elements.

23. The adaptive computing method of claim 18, wherein the first configuration information and the second configuration information are commingled with data to form a singular bit stream.

24. The adaptive computing method of claim 18, further comprising:

directing and scheduling the configuration of the plurality of heterogeneous computational elements for the first functional mode and the reconfiguration of the plurality of heterogeneous computational elements for the second functional mode.

25. The adaptive computing method of claim 18, further comprising:

timing and scheduling the configuration and reconfiguration of the plurality of heterogeneous computational elements with corresponding data.

26. The adaptive computing method of claim 18, further comprising:

selecting the first configuration information and the second configuration information from a singular bit stream containing data commingled with a plurality of configuration information.

27. The adaptive computing method of claim 18, further comprising:

storing in a memory the first configuration information and the second configuration information.

28. The adaptive computing method of claim 18, wherein the plurality of heterogeneous computational elements may be configured and reconfigured, through the interconnection network and in response to a plurality of configuration information, to implement a plurality of logic functions of a data flow graph.

29. The adaptive computing method of claim 18, wherein the interconnection network may be further configured to performa plurality of logic decisions of a data flow graph.

30. The adaptive computing method of claim 18, further comprising:

generating a plurality of control bits;

in response to the plurality of control bits, select an input line from the interconnection network for the reception of input information; and

in response to the plurality of control bits, selecting an output line from the interconnection network for the transfer of output information.

31. The adaptive computing method of claim 18, wherein the adaptive computing method is operable within a mobile terminal having a plurality of operating modes.

32. The adaptive computing method of claim 31, wherein the plurality of operating modes of the mobile terminal includes mobile telecommunication, personal digital assistance, multimedia reception, mobile packet-based communication, and paging.

33. An adaptive computing integrated circuit, comprising:

a plurality of reconfigurable matrices, the plurality of reconfigurable matrices including a plurality of heterogeneous computation units, each heterogeneous computation unit of the plurality of heterogeneous computation units formed from a selected configuration, of a plurality of configurations, of a plurality of fixed computational elements, the plurality of fixed computational elements including a first computational element having a first architecture and a second computational element having a second architecture, the first architecture distinct from the second architecture, the plurality of heterogeneous computation units coupled to an interconnect network and reconfigurable in response to configuration information; and

a matrix interconnection network coupled to the plurality of reconfigurable matrices, the matrix interconnection network operative to reconfigure the plurality of reconfigurable matrices in response to the configuration information for a plurality of operating modes.

34. The adaptive computing integrated circuit of claim 33, wherein each computation unit of the plurality of heterogeneous computation units is selectively reconfigurable and operative to execute a distinct algorithm of a plurality of algorithms.

35. The adaptive computing integrated circuit of claim 33, further comprising:

a controller coupled to the plurality of reconfigurable matrices, the controller operative to provide the configuration information to the reconfigurable matrices and to the matrix interconnection network.

36. The adaptive computing integrated circuit of claim 35, wherein the controller is further operative to detect and select the configuration information from a singular input bit stream of commingled data and configuration information.

37. The adaptive computing integrated circuit of claim 35, wherein the controller is embodied as a predetermined configuration of a reconfigurable matrix.

38. The adaptive computing integrated circuit of claim 35, wherein the controller is further operative to direct and schedule the configuration of the plurality of fixed computational elements for the plurality of operating modes.

39. The adaptive computing integrated circuit of claim 35, wherein the controller is further operative to time and schedule the configuration and reconfiguration of the plurality of fixed computational elements with corresponding data.

40. The adaptive computing integrated circuit of claim 35, further comprising:

a memory coupled to the controller and to the plurality of reconfigurable matrices, the memory operative to store the configuration information.

41. The adaptive computing integrated circuit of claim 40, wherein the memory is embodied as a predetermined configuration of a reconfigurable matrix.

42. The adaptive computing integrated circuit of claim 33, wherein the plurality of operating modes includes a first operating mode and a second operating mode, the first operating mode being different than the second operating mode.

43. The adaptive computing integrated circuit of claim 33, wherein the first architecture and the second architecture are selected from a plurality of specific architectures, the plurality of specific architectures including functions for memory, addition, multiplication, complex multiplication, subtraction, configuration, reconfiguration, control, input, output, and field programmability.

44. The adaptive computing integrated circuit of claim 33, wherein the plurality of operating modes includes linear algorithmic operations, non-linear algorithmic operations, finite state machine operations, memory operations, and bit-level manipulations.

45. The adaptive computing integrated circuit of claim 33, wherein the first architecture and the second architecture are selected to comparatively minimize power consumption of the adaptive computing integrated circuit.

46. The adaptive computing integrated circuit of claim 33, wherein an interconnection network portion of the matrix interconnection network reconfigurably routes data and control information between and among the plurality of fixed computational elements.

47. The adaptive computing integrated circuit of claim 33, wherein the configuration information is commingled with data to form a singular bit stream.

48. An adaptive computing integrated circuit, comprising:

a plurality of heterogeneous computational elements, the plurality of heterogeneous computational elements including a first computational element and a second computational element, the first computational element having a first fixed architecture and the second computational element having a second fixed architecture, the first fixed architecture being different than the second fixed architecture;

an interconnection network coupled to the plurality of heterogeneous computational elements, the interconnection network operative to configure the plurality of heterogeneous computational elements for a first functional mode of a plurality of functional modes, in response to first configuration information, and the interconnection network further operative to reconfigure the plurality of heterogeneous computational elements for a second functional mode of the plurality of functional modes, in response to second configuration information, the first functional mode being different than the second functional mode;

wherein a first subset of the plurality of heterogeneous computational elements is configured for a controller operating mode, the controller operating mode including functions for directing configuration and reconfiguration of the plurality of heterogeneous computational elements, for selecting the first configuration information and the second configuration information from a singular bit stream containing data commingled with a plurality of configuration information, and for scheduling the configuration and reconfiguration of the plurality of heterogeneous computational elements with corresponding data; and

wherein a second subset of the plurality of heterogeneous computational elements is configured for a memory operating mode for storing the first configuration information and the second configuration.

49. The adaptive computing integrated circuit of claim 48, wherein the first fixed architecture and the second fixed architecture are selected from a plurality of fixed architectures, the plurality of fixed architectures including functions for memory, addition, multiplication, complex multiplication, subtraction, configuration, reconfiguration, control, input, output, and field programmability.

50. The adaptive computing integrated circuit of claim 48, wherein the plurality of functional modes includes linear algorithmic operations, non-linear algorithmic operations, finite state machine operations, memory operations, and bit-level manipulations.

51. The adaptive computing integrated circuit of claim 48, wherein the adaptive computing integrated circuit is embodied within a mobile terminal having a plurality of operating modes.

52. The adaptive computing integrated circuit of claim 51, wherein the plurality of operating modes of the mobile terminal includes mobile telecommunication, personal digital assistance, multimedia reception, mobile packet-based communication, and paging.

53. An adaptive computing integrated circuit, comprising:

a plurality of heterogeneous computational elements, the plurality of heterogeneous computational elements including a first computational element and a second computational element, the first computational element having a first fixed architecture and the second computational element having a second fixed architecture of a plurality of fixed architectures, the first fixed architecture being different than the second fixed architecture, and the plurality of fixed architectures including functions for memory, addition, multiplication, complex multiplication, subtraction, configuration, reconfiguration, control, input, output, and field programmability; and

an interconnection network coupled to the plurality of heterogeneous computational elements, the interconnection network operative to configure the plurality of heterogeneous computational elements for a first functional mode of a plurality of functional modes, in response to first configuration information, and the interconnection network further operative to reconfigure the plurality of heterogeneous computational elements for a second functional mode of the plurality of functional modes, in response to second configuration information, the first functional mode being different than the second functional mode.

54. The adaptive computing integrated circuit of claim 53, wherein the plurality of functional modes includes linear algorithmic operations, non-linear algorithmic operations, finite state machine operations, memory operations, and bit-level manipulations.

55. The adaptive computing integrated circuit of claim 53, wherein the plurality of fixed architectures are selected to comparatively minimize power consumption of the adaptive computing integrated circuit.

56. The adaptive computing integrated circuit of claim 53, wherein the interconnection network reconfigurably routes data and control information between and among the plurality of heterogeneous computational elements.

57. The adaptive computing integrated circuit of claim 53, wherein the first configuration information and the second configuration information are commingled with data to form a singular bit stream.

58. The adaptive computing integrated circuit of claim 53, further comprising:

a controller coupled to the plurality of heterogeneous computational elements and to the interconnection network, the controller operative to direct and schedule the configuration of the plurality of heterogeneous computational elements for the first functional mode and the reconfiguration of the plurality of heterogeneous computational elements for the second functional mode.

59. The adaptive computing integrated circuit of claim 58, wherein the controller is further operative to time and schedule the configuration and reconfiguration of the plurality of heterogeneous computational elements with corresponding data.

60. The adaptive computing integrated circuit of claim 59, wherein the controller is further operative to select the first configuration information and the second configuration information from a singular bit stream containing data commingled with a plurality of configuration information.

61. The adaptive computing integrated circuit of claim 53, further comprising:

a memory coupled to the plurality of heterogeneous computational elements and to the interconnection network, the memory operative to store the first configuration information and the second configuration information.

62. The adaptive computing integrated circuit of claim 53, wherein the adaptive computing integrated circuit is embodied within a mobile terminal having a plurality of operating modes.

63. The adaptive computing integrated circuit of claim 62, wherein the plurality of operating modes of the mobile terminal includes mobile telecommunication, personal digital assistance, multimedia reception, mobile packet-based communication, and paging.

64. An adaptive computing integrated circuit, comprising:

a plurality of heterogeneous computational elements, the plurality of heterogeneous computational elements including a first computational element and a second computational element, the first computational element having a first fixed architecture and the second computational element having a second fixed architecture, the first fixed architecture being different than the second fixed architecture; and

an interconnection network coupled to the plurality of heterogeneous computational elements, the interconnection network operative to configure the plurality of heterogeneous computational elements for a first functional mode of a plurality of functional modes, in response to first configuration information, and the interconnection network further operative to reconfigure the plurality of heterogeneous computational elements for a second functional mode of the plurality of functional modes, in response to second configuration information, the first functional mode being different than the second functional mode, and the plurality of functional modes including linear algorithmic operations, non-linear algorithmic operations, finite state machine operations, memory operations, and bit-level manipulations.

65. The adaptive computing integrated circuit of claim 64, wherein the first fixed architecture and the second fixed architecture are selected from a plurality of specific architectures, the plurality of specific architectures including functions for memory, addition, multiplication, complex multiplication, subtraction, configuration, reconfiguration, control, input, output, and field programmability.

66. The adaptive computing integrated circuit of claim 64, wherein the first fixed architecture and the second fixed architecture are selected to comparatively minimize power consumption of the adaptive computing integrated circuit.

67. The adaptive computing integrated circuit of claim 64, wherein the interconnection network reconfigurably routes data and control information between and among the plurality of heterogeneous computational elements.

68. The adaptive computing integrated circuit of claim 64, wherein the first configuration information and the second configuration information are commingled with data to form a singular bit stream.

69. The adaptive computing integrated circuit of claim 64, further comprising:

a controller coupled to the plurality of heterogeneous computational elements and to the interconnection network, the controller operative to direct and schedule the configuration of the plurality of heterogeneous computational elements for the first functional mode and the reconfiguration of the plurality of heterogeneous computational elements for the second functional mode.

70. The adaptive computing integrated circuit of claim 69, wherein the controller is further operative to time and schedule the configuration and reconfiguration of the plurality of heterogeneous computational elements with corresponding data.

71. The adaptive computing integrated circuit of claim 69, wherein the controller is further operative to select the first configuration information and the second configuration information from a singular bit stream containing data commingled with a plurality of configuration information.

72. The adaptive computing integrated circuit of claim 64, further comprising:

a memory coupled to the plurality of heterogeneous computational elements and to the interconnection network, the memory operative to store the first configuration information and the second configuration information.

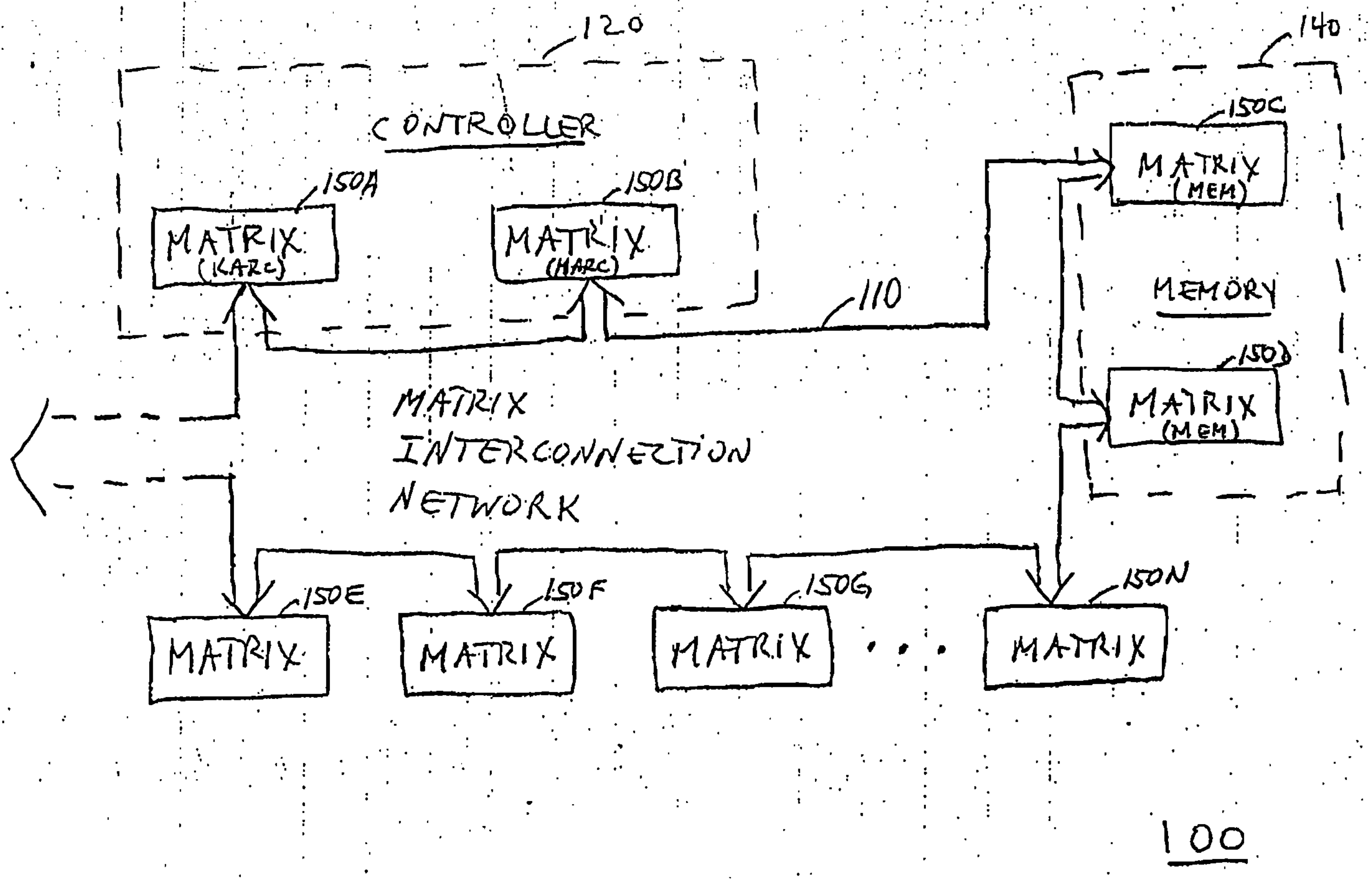
73. The adaptive computing integrated circuit of claim 64, wherein the adaptive computing integrated circuit is embodied within a mobile terminal having a plurality of operating modes.

74. The adaptive computing integrated circuit of claim 73, wherein the plurality of operating modes of the mobile terminal includes mobile telecommunication, personal digital assistance, multimedia reception, mobile packet-based communication, and paging.

ADAPTIVE INTEGRATED CIRCUITRY WITH HETEROGENEOUS AND RECONFIGURABLE MATRICES OF DIVERSE AND ADAPTIVE COMPUTATIONAL UNITS HAVING FIXED, APPLICATION SPECIFIC COMPUTATIONAL ELEMENTS

Abstract of the Disclosure

The present invention concerns a new category of integrated circuitry and a new methodology for adaptive or reconfigurable computing. The preferred IC embodiment includes a plurality of heterogeneous computational elements coupled to an interconnection network. The plurality of heterogeneous computational elements include corresponding computational elements having fixed and differing architectures, such as fixed architectures for different functions such as memory, addition, multiplication, complex multiplication, subtraction, configuration, reconfiguration, control, input, output, and field programmability. In response to configuration information, the interconnection network is operative in real-time to configure and reconfigure the plurality of heterogeneous computational elements for a plurality of different functional modes, including linear algorithmic operations, non-linear algorithmic operations, finite state machine operations, memory operations, and bit-level manipulations. The various fixed architectures are selected to comparatively minimize power consumption and increase performance of the adaptive computing integrated circuit, particularly suitable for mobile, hand-held or other battery-powered computing applications.



ADAPTIVE COMPUTING ENGINE (ACE)

FIG. 1

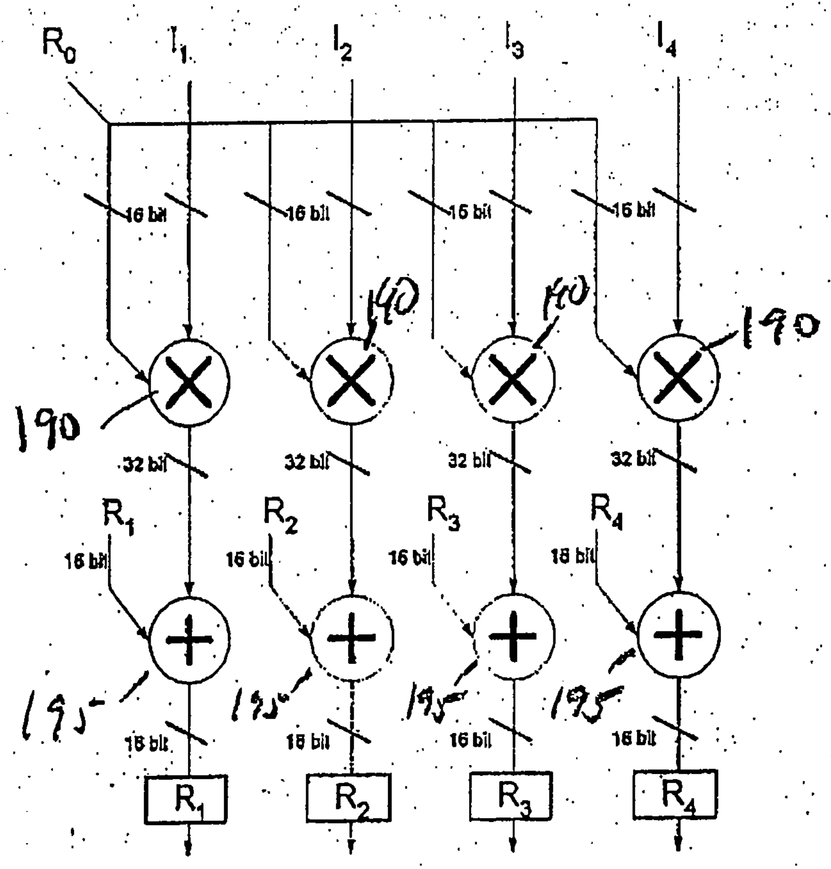

R0
I1
I2
I3
I4
16 bit
16 bit
16 bit
16 bit
190
140
140
190
32 bit
32 bit
32 bit
32 bit
R1
R2
R3
R4
16 bit
16 bit
16 bit
16 bit
195
195
195
195
16 bit
16 bit
16 bit
16 bit
R1
R2
R3
R4

FIG. 2

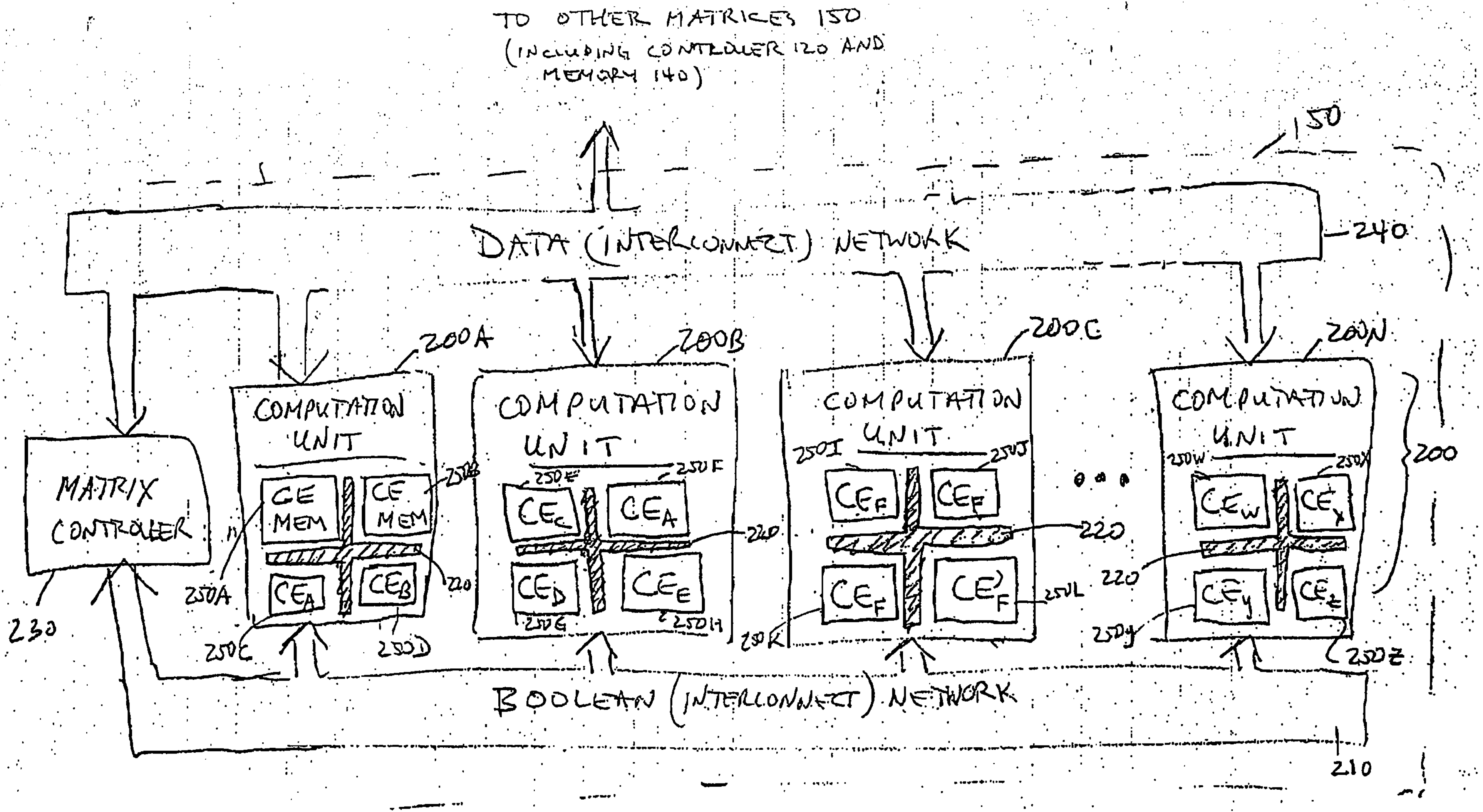

TO OTHER MATRICES 150
(INCLUDING CONTROLLER 120 AND
MEMORY 140)
150
DATA (INTERCONNECT) NETWORK
240
200A
200B
200C
200N
MATRIX
CONTROLLER
230
COMPUTATION
UNIT
CE
MEM
CE
MEM
250A
250B
CE$_A$
CE$_B$
250C
250D
220
COMPUTATION
UNIT
250E
250F
CE$_C$
CE$_A$
CE$_D$
CE$_E$
250G
250H
220
COMPUTATION
UNIT
250I
250J
CE$_F$
CE$_F$
CE$_F$
CE$'_F$
250K
250L
220
• • •
COMPUTATION
UNIT
250W
250X
CE$_W$
CE$_X$
CE$_Y$
CE$_Z$
250Y
250Z
220
200
BOOLEAN (INTERCONNECT) NETWORK
210

FIG. 3

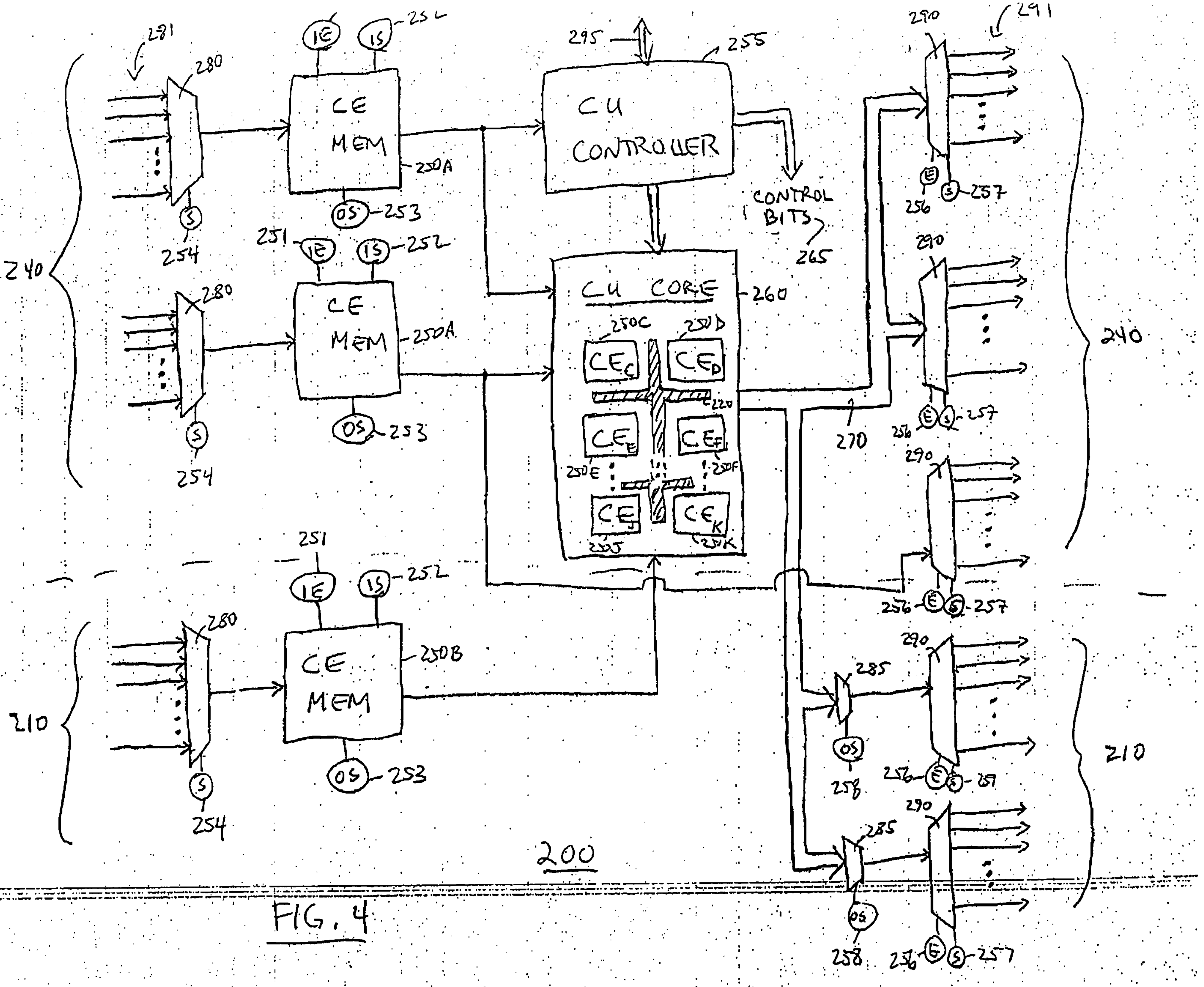

281
280
CE MEM
250A
IE
IS
252
253
OS
S
254
251
240
CU CONTROLLER
255
295
CONTROL BITS
265
CU CORE
260
250C
250D
CE_C
CE_D
220
CE_E
CE_F
250E
250F
CE_J
CE_K
250J
250K
270
290
291
E
256
257
285
258
250B
210
200

FIG. 4

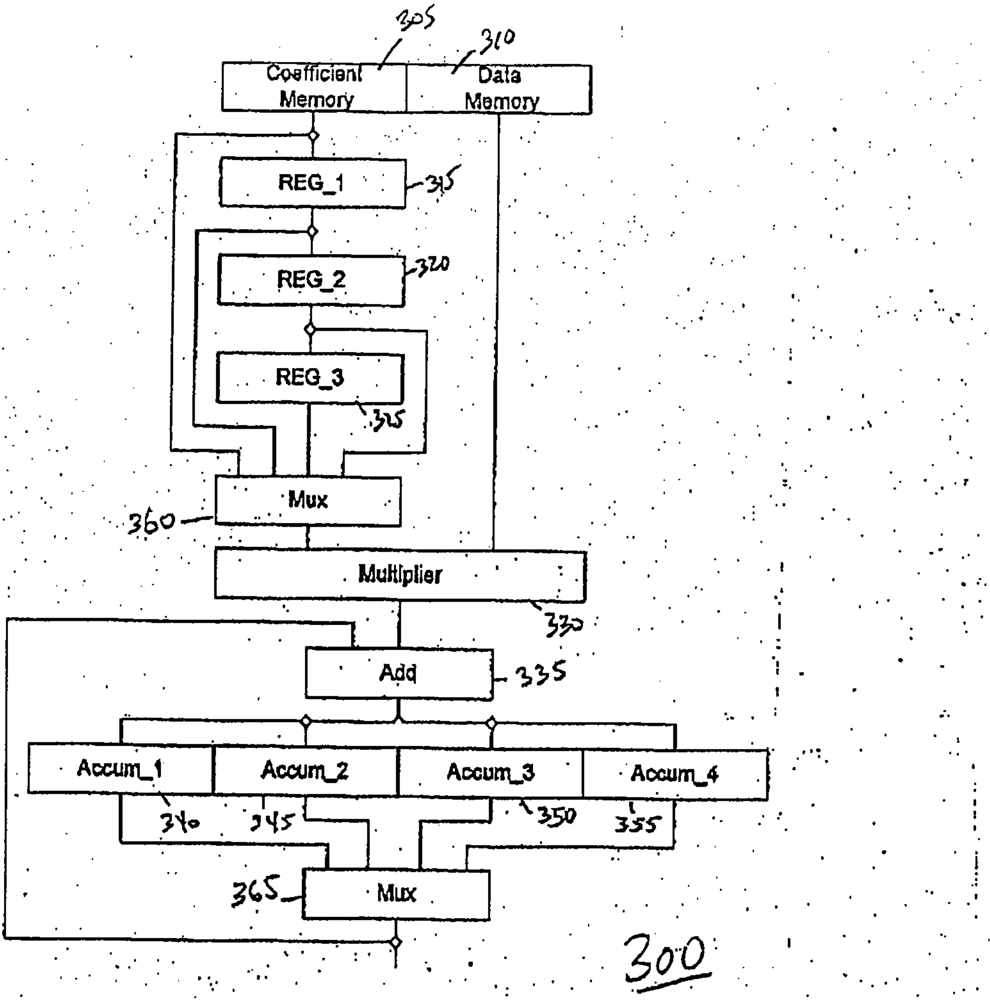
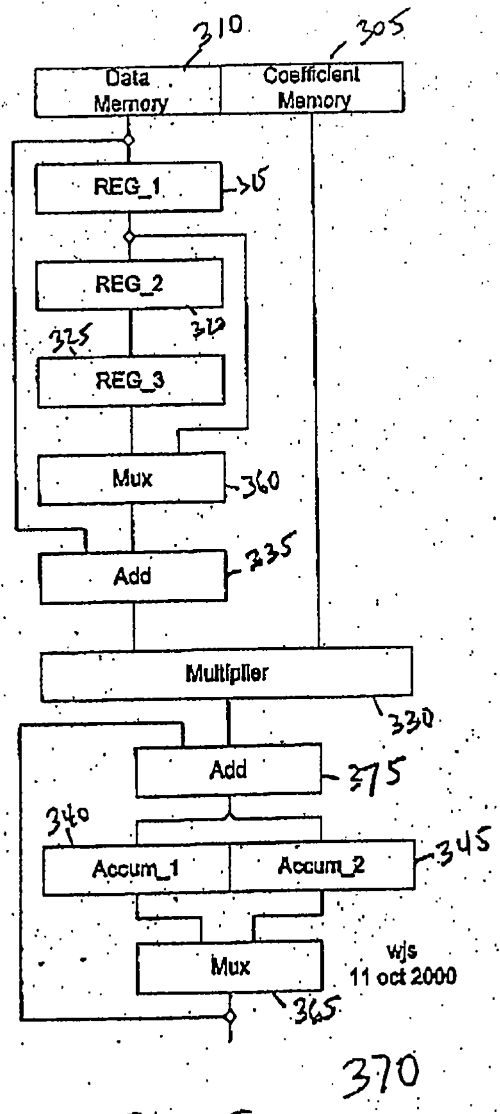
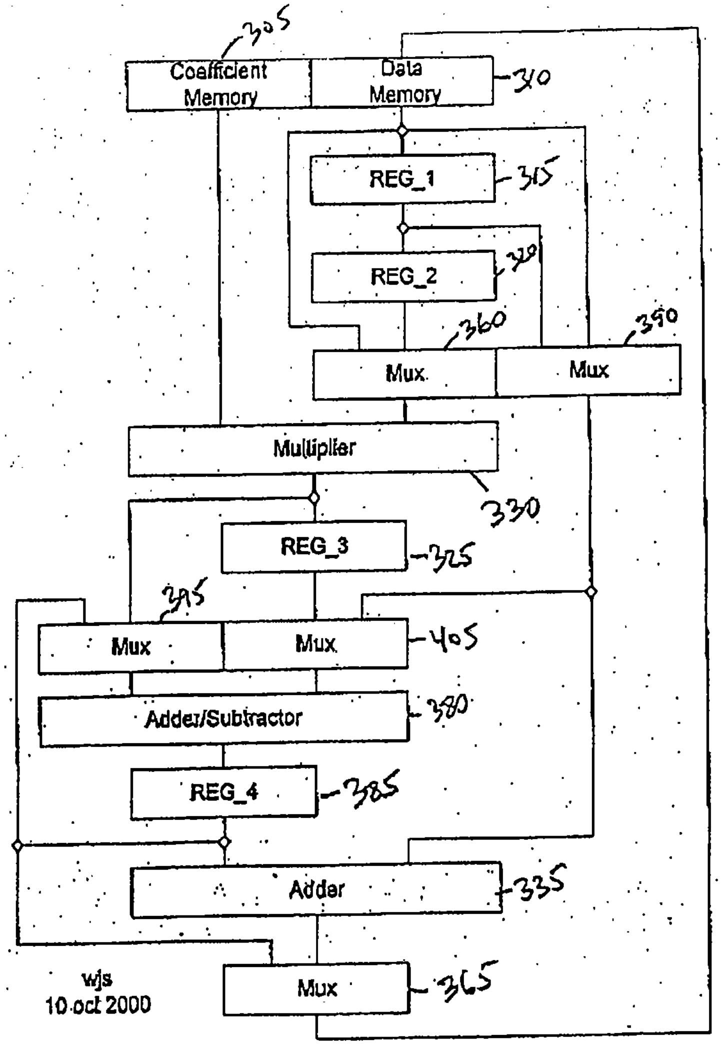

305
310
Coefficient Memory
Data Memory
REG_1
315
REG_2
320
REG_3
325
Mux
360
Multiplier
330
Add
335
Accum_1
Accum_2
Accum_3
Accum_4
340
345
350
355
Mux
365
300

FIG. 5A



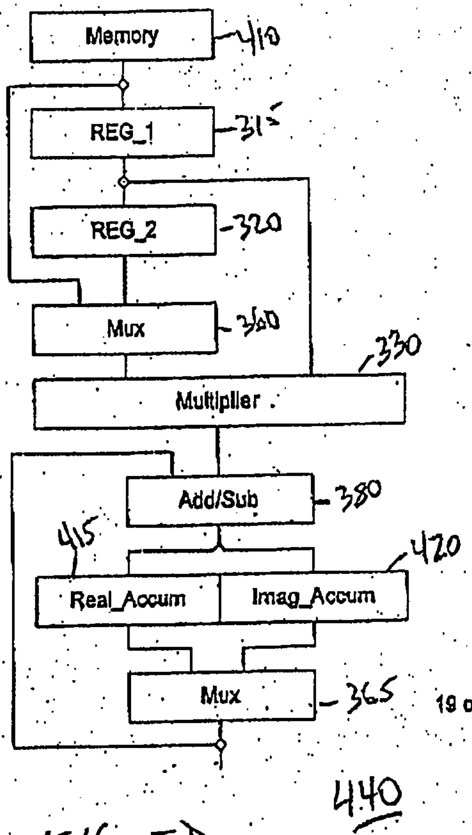
wjs
11 oct 2000

FIG. 5B



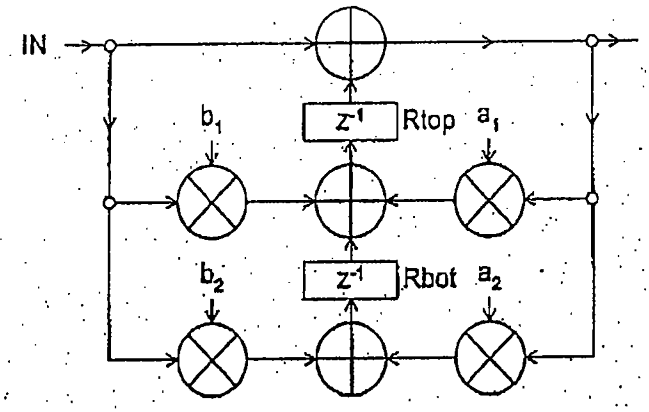
wjs
10 oct 2000

FIG. 5C



wjs
19 oct 2000

FIG. 5D



OUT <-- IN + Rtop

Rbot <-- IN * b_2 + OUT * a_2

Rtop <-- IN * b_1 + OUT * a_1 + Rbot

wjs
23 oct 2000

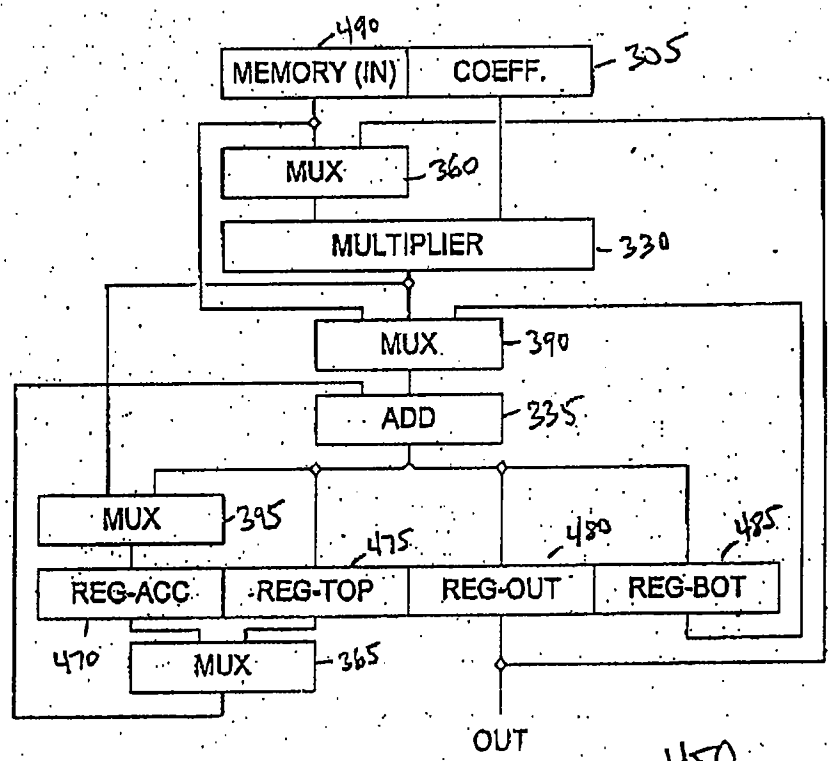
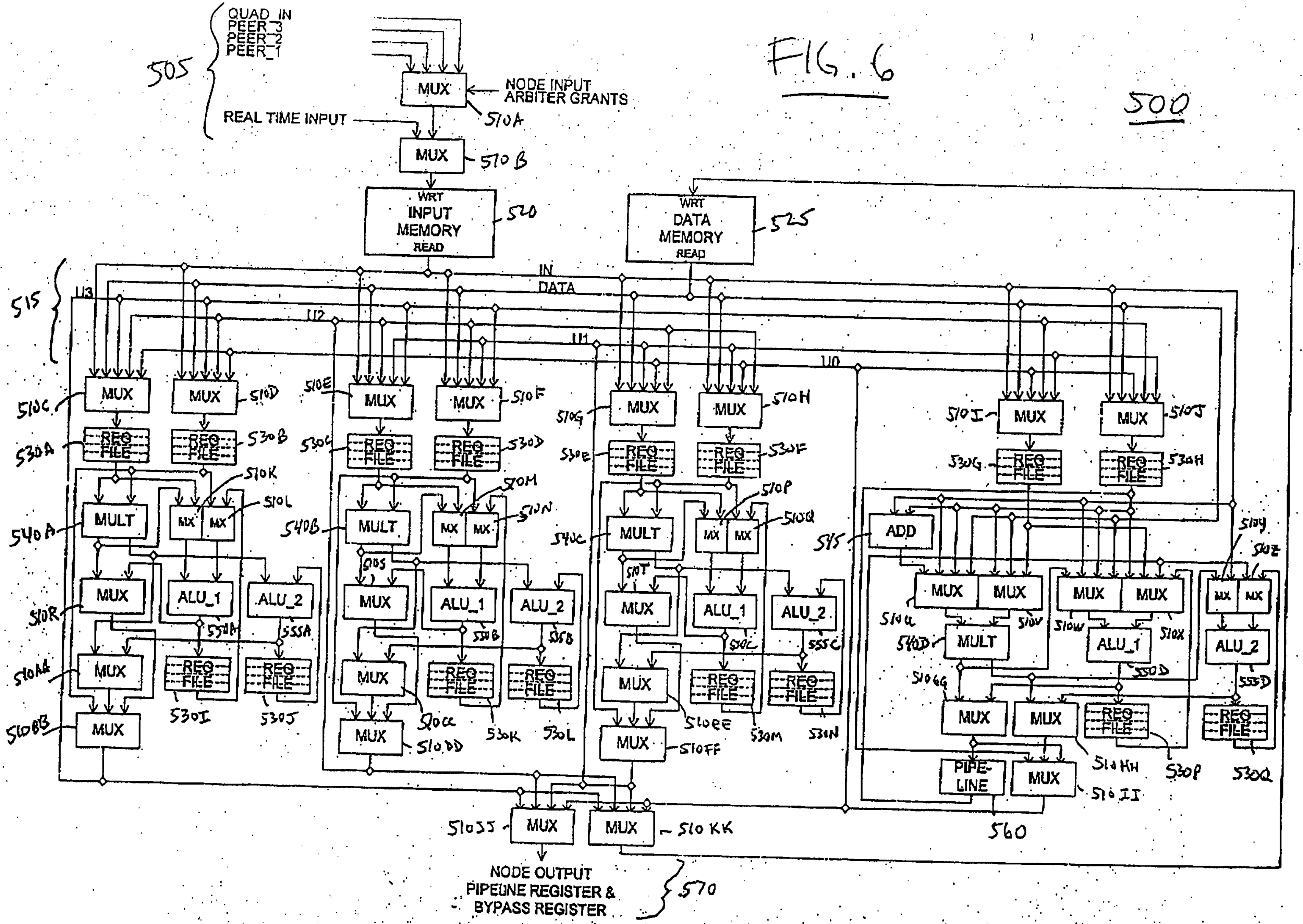


FIG. 5E

FIG. 6


500
505
QUAD_IN
PEER_3
PEER_2
PEER_1
MUX
NODE INPUT ARBITER GRANTS
510A
REAL TIME INPUT
MUX
510B
WRT
INPUT MEMORY
READ
520
WRT
DATA MEMORY
READ
525
515
IN
DATA
U3
U2
U1
U0
MUX
510C
MUX
510D
510E
MUX
MUX
510F
510G
MUX
MUX
510H
510I
MUX
MUX
510J
530A
REG FILE
REG FILE
530B
530C
REG FILE
REG FILE
530D
530E
REG FILE
REG FILE
530F
530G
REG FILE
REG FILE
530H
510K
510L
540A
MULT
MX
MX
540B
MULT
MX
MX
510M
510N
540C
MULT
MX
MX
510P
510Q
545
ADD
510Y
510Z
MUX
ALU_1
ALU_2
510R
550A
555A
510S
MUX
ALU_1
ALU_2
550B
555B
510T
MUX
ALU_1
ALU_2
550C
555C
MUX
MUX
MUX
MUX
MX
MX
510U
510V
510W
510X
540D
MULT
ALU_1
ALU_2
550D
555D
510AA
MUX
REG FILE
REG FILE
MUX
REG FILE
REG FILE
510CC
MUX
REG FILE
REG FILE
510EE
510GG
MUX
MUX
REG FILE
REG FILE
530I
530J
530K
530L
530M
530N
510HH
530P
530Q
510BB
MUX
MUX
510DD
MUX
510FF
PIPE-LINE
MUX
510II
560
510JJ
MUX
MUX
510KK
NODE OUTPUT
PIPELINE REGISTER &
BYPASS REGISTER
570

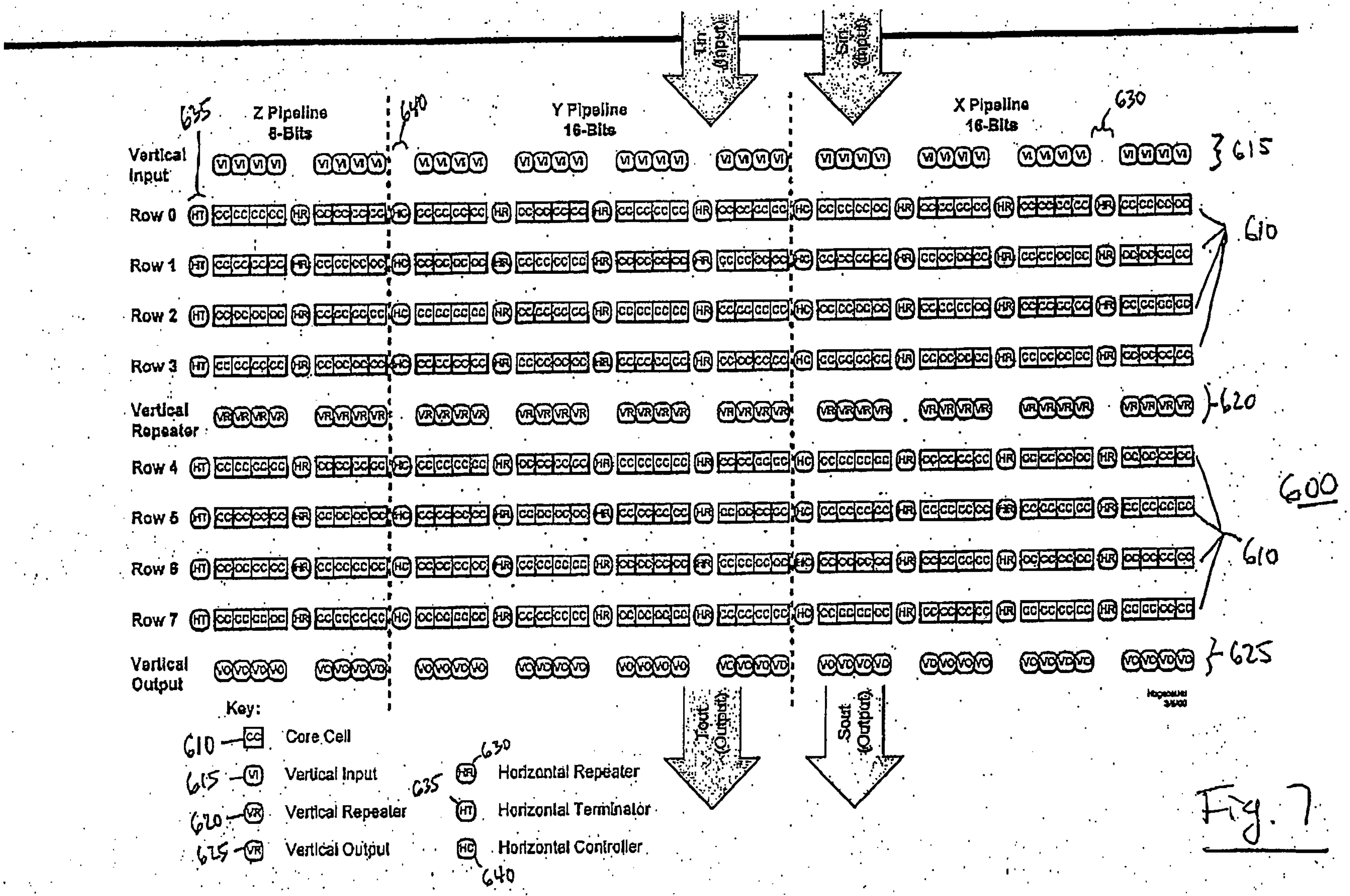

Tin (Input)
Sin (Input)
635
Z Pipeline
8-Bits
640
Y Pipeline
16-Bits
X Pipeline
16-Bits
630
Vertical Input
615
Row 0
Row 1
Row 2
Row 3
610
Vertical Repeater
620
Row 4
Row 5
Row 6
Row 7
600
610
Vertical Output
625
VI
CC
HT
HR
HC
VR
VO
Tout (Output)
Sout (Output)
Key:
610 CC Core Cell
615 VI Vertical Input
620 VR Vertical Repeater
625 VR Vertical Output
630 HR Horizontal Repeater
635 HT Horizontal Terminator
640 HC Horizontal Controller

Fig. 7

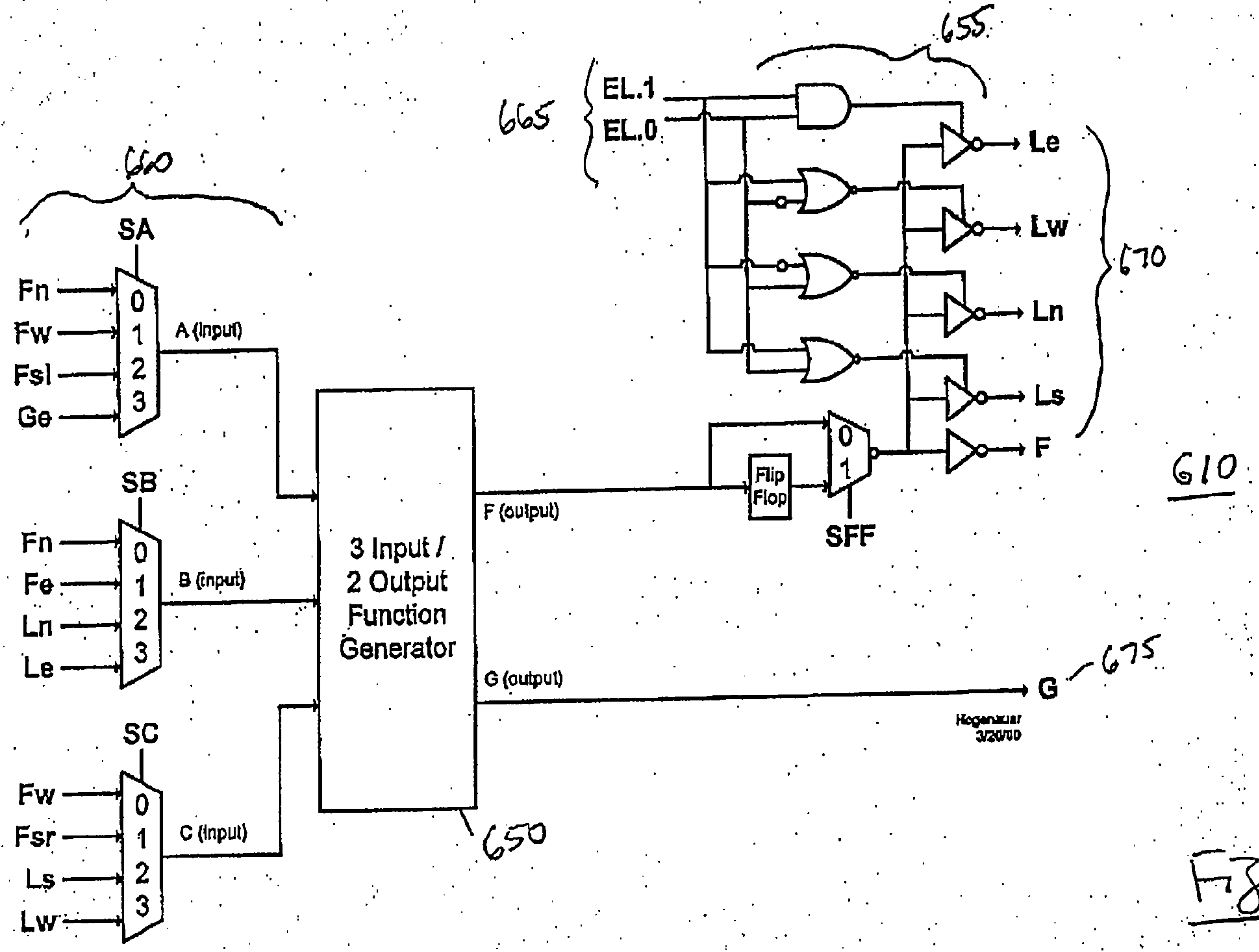

660
SA
Fn
Fw
Fsl
Ge
0
1
2
3
A (input)
SB
Fn
Fe
Ln
Le
0
1
2
3
B (input)
SC
Fw
Fsr
Ls
Lw
0
1
2
3
C (input)
3 Input / 2 Output Function Generator
650
F (output)
G (output)
665
EL.1
EL.0
655
Flip Flop
0
1
SFF
Le
Lw
Ln
Ls
F
670
610
G
675
Hogenauer 3/20/00

Fig. 8

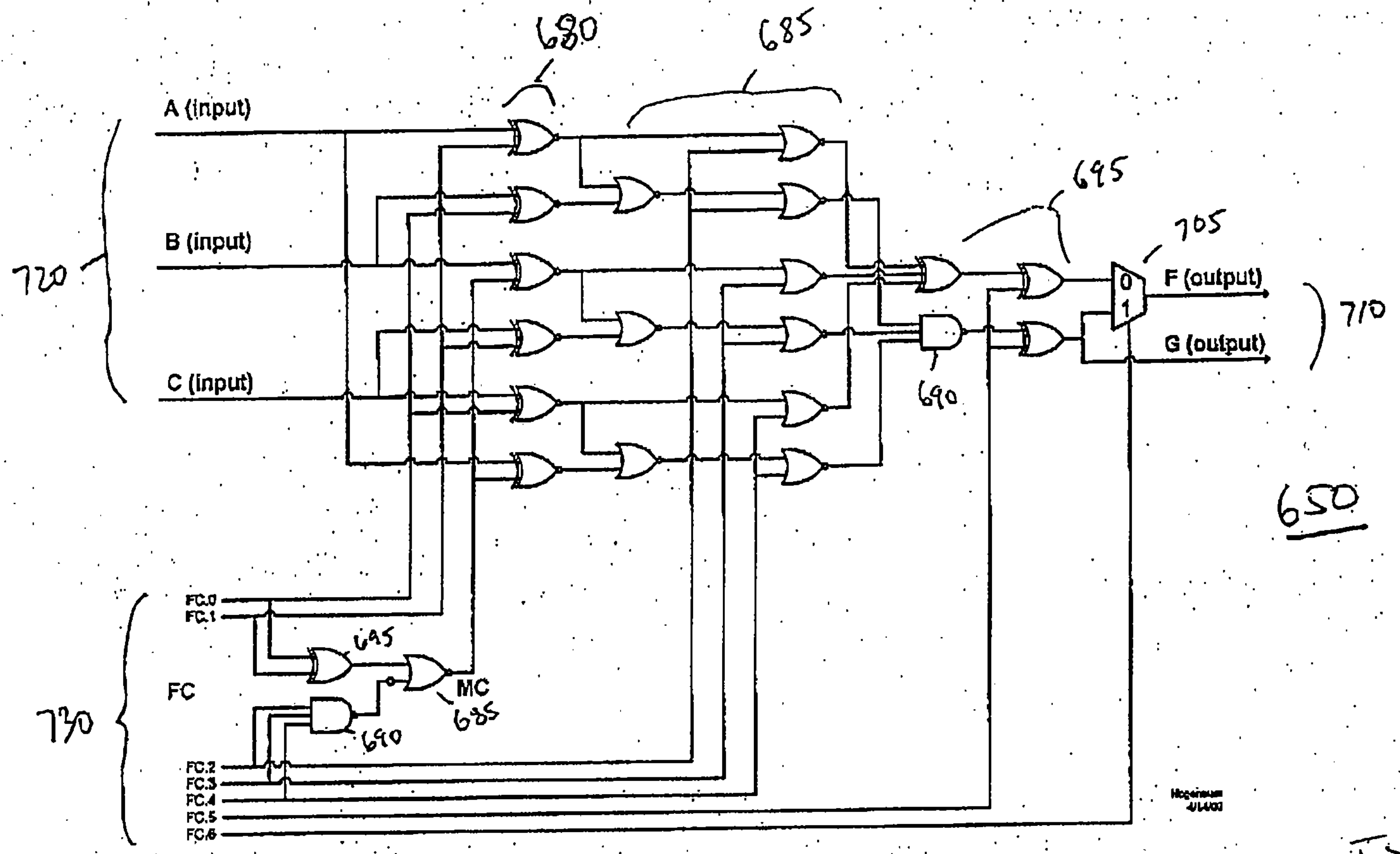

680
685
A (input)
B (input)
C (input)
720
695
705
F (output)
G (output)
710
0
1
690
650
FC.0
FC.1
695
MC
685
FC
690
730
FC.2
FC.3
FC.4
FC.5
FC.6

Fig. 9

Northstar Chip
Architecture Definition

PSN Node
Programmable Scalar Node

Revision 1.36

Quicksilver Confidential

Author: Dan Chuang

Table of Contents

Tables

Figures

1. PSN Node

The Programmable Scalar Node (PSN) node or ARC node is a node based on the ARC 32-bit RISC processor. The method of integrating the ARC core into the MIN network will adhere strictly to the homogeneous node wrapper / heterogeneous node concept. This document is meant to describe the architecture from a nodal perspective. Any system level requirements on the PSN or K-node will be addressed elsewhere. Figure 1 shows the PSN node in the context of the Node Wrapper and network input and output. Figure 2 shows the high-level block diagram of the PSN Core.

Figure 1: PSN Nodal Architecture

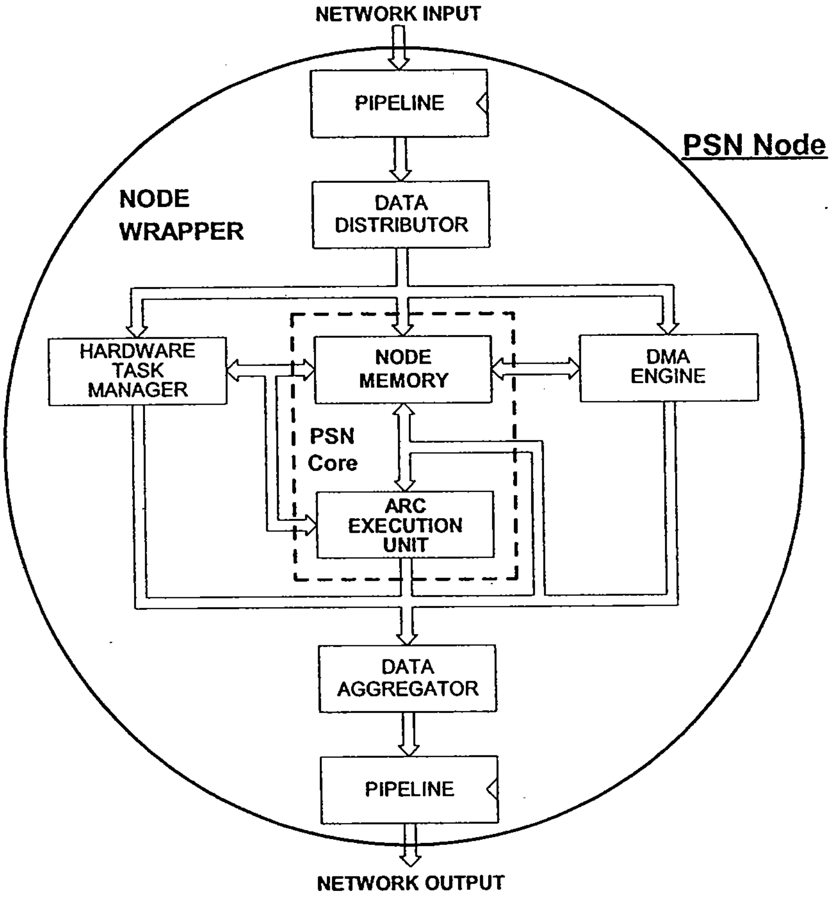
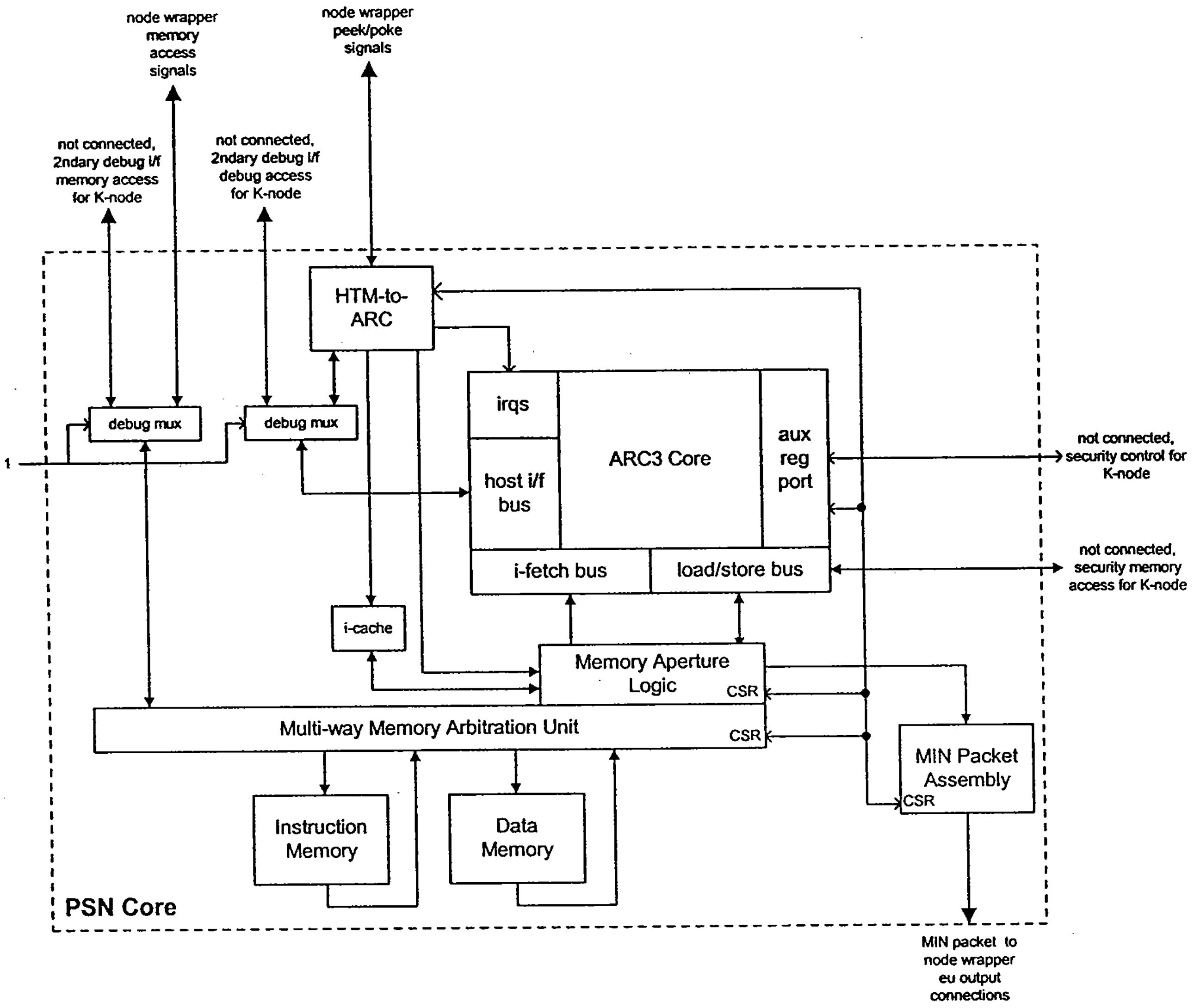


Figure 2: PSN Node Block Diagram


node wrapper memory access signals
node wrapper peek/poke signals
not connected, 2ndary debug I/f memory access for K-node
not connected, 2ndary debug I/f debug access for K-node
HTM-to-ARC
irqs
ARC3 Core
aux reg port
host i/f bus
i-fetch bus
load/store bus
debug mux
debug mux
1
not connected, security control for K-node
not connected, security memory access for K-node
i-cache
Memory Aperture Logic
CSR
Multi-way Memory Arbitration Unit
CSR
MIN Packet Assembly
CSR
Instruction Memory
Data Memory
PSN Core
MIN packet to node wrapper eu output connections

The PSN node has two flavors in the ACM system: as the kernel node that runs the operating system for the ACM and as a general purpose RISC node. As a general purpose RISC node, the PSN is typically used for decision intensive applications requiring large code space. When the PSN node is used as the kernel node, it is also called the K-node or K-ARC node. Essentially, both K-node and PSN use the same PSN Core and the same Node Wrapper to connect to the MIN. However, in the case of a K-node, the security bit in the Secure Configuration Register shown in Table 1 is set by default. This bit would determine whether the MIN words sent out by the MIN Packet Assembly block has the security bit set and whether it can produce the full range of MIN service words. Any other K-node specific differences from the PSN will be discussed in Section 2: K-node Specific Features.

Table 1: Security Configuration Register in Node Specific Space

Address	Name	Function

Address	Name	Function
0x1000	SCR	Determines the security status of a node, 1-bit PSN default: 0, K-node default: 1

This node specific register resides physically in the MIN Packet Assembly block described in the later section.

1.1 ARC CPU

In the Northstar, the same ARC3 core will be used for both the K-node and the PSN with only a few exceptions. The ARC is a 32-bit little-endian RISC processor with 4 stage pipeline. Throughout this document, any mentioning of "word" means a 16-bit word, except in the case of a MIN word, which means the 51-bit MIN communication structure. The mentioning of "dword" or longword means a 32-bit word.

1.1.1 Instruction Set Architecture

- 3 Port Synchronous RAM Cell Register File (32x32)
- Fast Load Returns
- Fast 32x32 Barrel Shifter
- Fast 32x32 Multiplier
- Swap
- Min/Max
- Normalize

1.1.2 Memory Configuration

- 1kB Direct Mapped Standard I-Cache, with line length of 16 instruction words, instruction bus width extended to 26-bit for addressing up to 64MB of instructions
- Load/store memory bus size extended to 28-bit for addressing up to 256MB of data
- 2kB Local Load Store RAM access for Secure Memory and ROM (K-node only, unconnected in PSN)

1.1.3 Other Features

- Timer (two 32-bit programmable timers, default to disabled)
- Interrupt controller (2-level priority mask-able, extended to accommodate more interrupts, 9 total)
- Clock gating (shuts down the clock tree to the ARC pipeline when it is halted or in sleep)

1.1.4 Programmers Model, Summary

Table 2: Instruction Set: Single Operand

Code	Instruction	Notes
03,[0]	FLAG	Set the flags
03,[3F,0]	BRK	Break Point
03,[3F,1]	SLEEP	Sleep

Code	Instruction	Notes
03,[1]	ASR	Arithmetic Shift Right by one
03,[2]	LSR	Logical Shift Right by one
03,[3]	ROR	Rotate Right by one
03,[4]	RRC	Rotate Right through Carry by one
03,[5]	SEXB	Sign Extend byte to long word
03,[6]	SEXW	Sign Extend word to long word
03,[7]	EXTB	Zero Extend byte to long word
03,[8]	EXTW	Zero Extend word to long word
03,[9]	SWAP	Swap words
03,[A]	NORM	Calculate shift value to normalize

Table 3: Instruction Set: Dual Operand

Code	Instruction	Notes
00	LD r + r	Delayed load (core registers only)
01,[10]	LD r + o, LR	Delayed load or load from aux. reg.
02,[10]	ST r + o, SR	Buffered store or store to aux. reg.
04	Bcc	Branch conditionally
05	BLcc	Branch and link conditionally
06	LPcc	Loop set up or jump conditionally
07	Jcc	Jump (and link) conditionally
08	ADD	Addition
09	ADC	Addition with carry
0A	SUB	Subtract
0B	SBC	Subtract with carry
0C	AND	Logical bit wise AND
0D	OR	Logical bit wise OR
0E	BIC	Logical bit wise AND with invert
0F	XOR	Logical bit wise exclusive-OR
10	ASL	Multiple arithmetic shift left
11	LSR	Multiple logical shift left
12	ASR	Multiple arithmetic shift right
13	ROR	Multiple rotate right
14	MUL64	32 x 32 signed multiply
15	MULU64	32 x 32 unsigned multiply
16	---	
17	---	
18	---	
19	---	
1A	---	
1B	---	
1C	---	
1D	---	
1E	MAX	Return larger of two signed integers
1F	MIN	Return smaller of two signed integers

The following tables summarize the core and auxiliary registers available in the PSN. The "SAVE?" column indicates whether the content of a register should be stored when a task is being de-scheduled while full context frame is required.

Table 4: Core Registers

Number	Name	Function	Save?
r0 - r28	Basecase	General Purpose Register	y
r29	ILINK1	Maskable interrupt register 1	

Number	Name	Function	Save?
r30	ILINK2	Maskable interrupt register 2	y
r31	BLINK	Branch link register	y
r32 - r56	---		
r57	MLO	Multiply Result: low 32 bits	y
r58	MMID	Multiply Result: middle 32 bits	
r59	MHI	Multiply Result: high 32 bits	y
r60	LP_COUNT	Loop Counter register (24-bit)	y
r61	SHIMMF	Short imm. data (setting flags)	
r62	LIMM	Long immediate data	
r63	SHIMM	Short imm. data (not setting flags)	

Table 5: Auxiliary Registers

Number	Name	Function	Save?
0x0	STATUS	Status Register	
0x1	SEMAPHORE	Host semaphore register	
0x2	LP_START	Loop Start Address	y
0x3	LP_END	Loop End Address	y
0x4	IDENTITY	ARC Identification register, Current Node's ID	
0x5	DEBUG	Debug register	
0x6 - 0xF	---	---	
0x10	IVIC	Cache invalidate register	
0x11	CHE_MODE_CTL	Mode bits for cache controller	
0x12	MULHI	High part of Multiply	
0x13 - 0x17	---	---	
0x18	LOCAL_RAM	Base Address of Local RAM (ROM + Secure RAM)	
0x19 - 0x20	---	---	
0x21	T0_COUNT	Timer0 count value, 32-bit, 0x0 on reset	
0x22	T0_CONTROL	Timer0 control register, 0x0 on reset	
0x23	T0_LIMIT	Timer0 counting limit value, default: 0x00FFFFFF	
0x24 - 0x60	---	---	
0x61	T1_COUNT	Timer1 count value, 32-bit, 0x0 on reset	
0x62	T1_CONTROL	Timer1 control register, 0x0 on reset	
0x63	T1_LIMIT	Timer1 counting limit value, default: 0x00FFFFFF	
0x64 - 0x72	---	---	
0x73	MADI_BUILD	Build: Multiple ARC Debug I/f	
0x74	LDSTRAM_BUILD	Build: LD/ST RAM	
0x75	TIMER_BUILD	Build: Timer	
0x76	AP_BUILD	Build: Actionpoints	
0x77	CACHE_BUILD	Build: I-Cache	
0x78	ADDSUB_BUILD	Build: Saturated Add/Sub	
0x79	DSPRAM_BUILD	Build: Scratch RAM & XY Memory	
0x7A	MAC_BUILD	Build: Mul MAC	
0x7B	MULTIPLY_BUILD	Build: Multiply	
0x7C	SWAP_BUILD	Build: Swap	
0x7D	NORM_BUILD	Build: Normalize	
0x7E	MINMAC_BUILD	Build: Min/Max	
0x7F	BARREL_BUILD	Build: Barrel Shift	
0x80	SHA_CSR	Control and status register for SHA-1 hardware	
0x81	BOOT_SEL	Boot_sel pin state	
0x82 - 0xFF	---	---	
0x100	MALB	Memory access local base address	
0x101	MALS	Memory access local page size	
0x102	MABB	Memory access bulk base address	
0x103	MABS	Memory access bulk page size	
0x104	MAEB	Memory access ext. base address	

Number	Name	Function	Save?
0x105	MAES	Memory access ext. page size	
0x106 - 0x107	---	Reserved	
0x108	MASP	Packed MAS parameters (Write only)	
0x109 - 0x11F	---	Reserved	
0x120	TASK_CONTINUE	Registered EU_CONTINUE signal (Read only)	
0x121	TASK_TEARDOWN	Registered EU_TEARDOWN signal (Read only)	
0x122	TASK_DONE	Generates the EU_DONE signal for the HTM	
0x123	---	Reserved	
0x124	MSG_RD	Read queued message received from MIN	
0x125 - 0x126	---	Reserved	
0x127	TPLP	TPL pointer table entry number	y
0x128	TASK_CFG	Packed Module pointer, Execution Env	
0x129	TEE	Task execution environment	
0x12A	MOD_PTR	Module pointer	
0x12B - 0x13F	---	Reserved	
0x140 - 0x15F	IPORTx_SIGN	Port counter sign of input port x (0-31)	
0x160 - 0x17F	OPORTx_SIGN	Port counter sign of output port x (0-31)	
0x180	INBUF0_P0	1st packed parameter word for input buffer 0	
0x181	INBUF0_P1	2nd packed parameter word for input buffer 0	
0x182	INBUF1_P0	1st packed parameter word for input buffer 1	
0x183	INBUF1_P1	2nd packed parameter word for input buffer 1	
0x184	INBUF2_P0	1st packed parameter word for input buffer 2	
0x185	INBUF2_P1	2nd packed parameter word for input buffer 2	
:	:	:	
0x190	INBUF8_P0	1st packed parameter word for input buffer 8	
0x191	INBUF8_P1	2nd packed parameter word for input buffer 8	
:	:	:	
0x1A0	INBUF16_P0	1st packed parameter word for input buffer 16	
0x1A1	INBUF16_P1	2nd packed parameter word for input buffer 16	
:	:	:	
0x1B0	INBUF30_P0	1st packed parameter word for input buffer 24	
0x1B1	INBUF30_P1	2nd packed parameter word for input buffer 24	
:	:	:	
0x1BE	INBUF31_P0	1st packed parameter word for input buffer 31	
0x1BF	INBUF31_P1	2nd packed parameter word for input buffer 31	
0x1C0	OUTBUF0_P	Packed parameter word for output buffer 0	
0x1C1	OUTBUF1_P	Packed parameter word for output buffer 1	
0x1C2	OUTBUF2_P	Packed parameter word for output buffer 2	
:	:	:	
0x1CF	OUTBUF15_P	Packed parameter word for output buffer 15	
0x1D0	OUTBUF16_P	Packed parameter word for output buffer 16	
:	:	:	
0x1DE	OUTBUF30_P	Packed parameter word for output buffer 30	
0x1DF	OUTBUF31_P	Packed parameter word for output buffer 31	
0x1E0 - 0x1EF	---	Reserved	
0x1F0	IBUF_CFG_MODE	Input buffer address pointer update mode	
0x1F1	PTP_PACKET	Send Point-to-Point Packet Mode header	
0x1F2	MIN_WR_LSP	Lower 32-bit of MIN word to send	
0x1F3	MIN_WR_MSP	Upper 19-bit of MIN word to send	
0x1F4	IBUF_EN	Number of input buffers / buffer enable bits (32-bit)	
0x1F5	OBUF_EN	Number of output buffers / buffer enable bits (32-bit)	
0x1F6	TASK_GO	Modifies this task's GO bit in the HTM	
0x1F7	MSG_SND	Send message to K-node	
0x1F8 - 0x1FF	---	Reserved	

The above tables listed only the register number with respect to the program running on the ARC. For MIN access via the ARC host port, the registers are mapped to the address as described by Table 17: Node I/O Address Map to ARC Host Access. In addition, the following table summarizes the node specific registers that are exposed only to K-node peek/poke accesses.

Table 6: Node Specific Registers

Address	Name	Function
0x1000	SCR	Security configuration of the PSN core
0x1008	MSG_MB	Message mailbox for K-node to peek, 3-bit message.
0x1010	MACR	Memory aggregation configuration, 3-bit, default: 0x1
0x1020	MAS_LOCK	Memory access lock, 1-bit, default: 0, not lock

1.2 Functional Description

1.2.1 Memory System

The 16kB of memory allocated to PSN plus the possible 2-tile memory aggregation mode for a total of 32kB can be setup, at initialization, to be 4kB-to-24kB instruction memory and 4kB-to-24kB data memory combinations depending on the application need. The range of these size combinations (shown in Table 8) can be reduced depending on actual circuit implementation result and timing consideration. Regardless, the instruction and data memories are accessed separate and independently by the ifetch and load/store buses respectively. The read-path structure of the memory aggregation scheme is shown in Figure 3; the write-path structure is similarly constructed, with the addition of the write_enable multiplexers similar to the address multiplexers and the addition of the data inputs to the memories connected to the load/store write data signal. MA[0] (Memory Aggregation register value from the Node Wrapper) determines how the address from ifetch and load/store buses are rearranged. A 2-bit configuration register is used to configure the multiplexers (Table 7), where the default value is 0x0 meaning, in 1-tile no aggregation mode, 12kB of instruction memory and 4kB of data memory, and in 2-tile aggregated mode, 24kB of instruction memory and 8kB for data memory. The sizes for the instruction and the data memories are configured in the increment of 4kB or 8kB as described in Table 8.

Figure 3: Node Memory Aggregation System

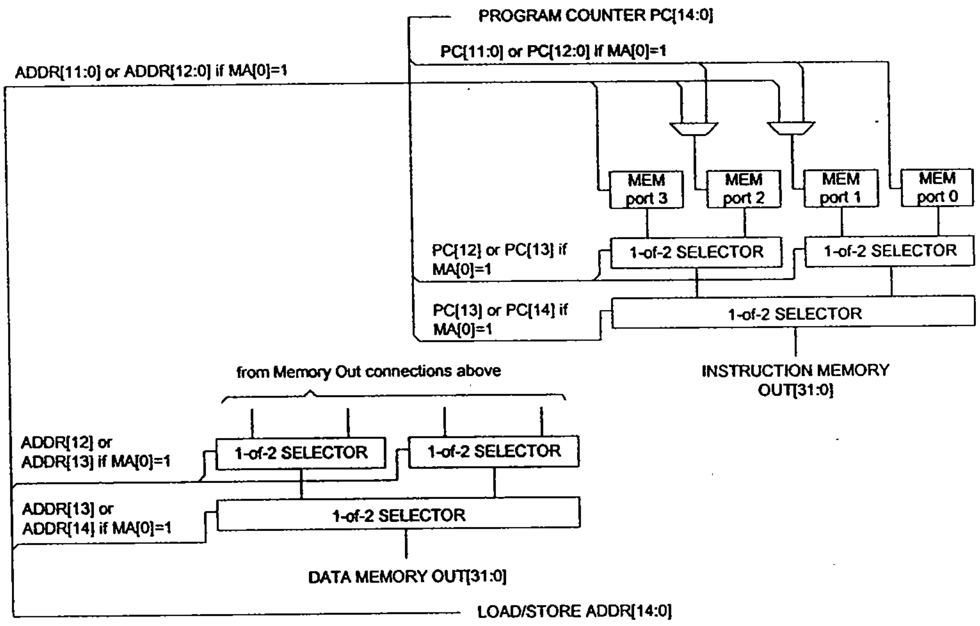

PROGRAM COUNTER PC[14:0]
PC[11:0] or PC[12:0] if MA[0]=1
ADDR[11:0] or ADDR[12:0] if MA[0]=1
MEM port 3
MEM port 2
MEM port 1
MEM port 0
PC[12] or PC[13] if MA[0]=1
1-of-2 SELECTOR
1-of-2 SELECTOR
PC[13] or PC[14] if MA[0]=1
1-of-2 SELECTOR
INSTRUCTION MEMORY OUT[31:0]
from Memory Out connections above
ADDR[12] or ADDR[13] if MA[0]=1
1-of-2 SELECTOR
1-of-2 SELECTOR
ADDR[13] or ADDR[14] if MA[0]=1
1-of-2 SELECTOR
DATA MEMORY OUT[31:0]
LOAD/STORE ADDR[14:0]

Table 7: Memory Aggregation Configuration Register in Node Specific Space

Address	Name	Function
0x1010	MACR	Memory aggregation configuration, 2-bit, default: 0x0

Table 8: Memory Aggregation Configuration

1-tile No Aggregation Mode		
MACR	Instruction Memory (kB)	Data Memory (kB)
00	12	4
01	8	8
11	4	12
2-tile Aggregation Mode		
MACR	Instruction Memory (kB)	Data Memory (kB)
00	24	8
01	16	16
11	8	24

This node specific register resides physically in the Multi-way Memory Arbitration Unit described in the following section.

1.2.2 Multi-way Memory Arbitration Unit (MMAU)

PSN will not be using ARC's memory subsystem (memory arbitration unit and sequencer) at all. The load/store memory access in particular is done similar to what ARC calls "local memory" access; and the ifetch access bypasses the cache logic and connects directly to the node memory. Therefore, there is no latency involved in accessing to/from the nodal memory, except in the case that the Node Wrapper is busy with a memory, the ARC will be hold off on any memory request.

The Multi-way Memory Arbitration Unit has possible three memory clients to serve, i-fetch, load/store, and memory access from the node wrapper. Its main function is to:

- Arbitrate between ARC access and Node Wrapper access to local node memory, meaning, granting and halting these memory client requests as necessary

The memory clients may want to access a memory source at the same time, and only one client can be accommodated for one physical memory source. The unit is called "multi-way" because there are several memory sources involved here, i.e. local instruction and data memories, each with individual read and write ports. Essentially, there are three independent arbiters in this block. Each arbiter has different set of memory clients to service. In general, a simple priority arbitration scheme is used as follows, the priority given from highest to lowest, where Node Wrapper access always wins:

Read Arbiter i

1. Node wrapper read
2. ARC ifetch

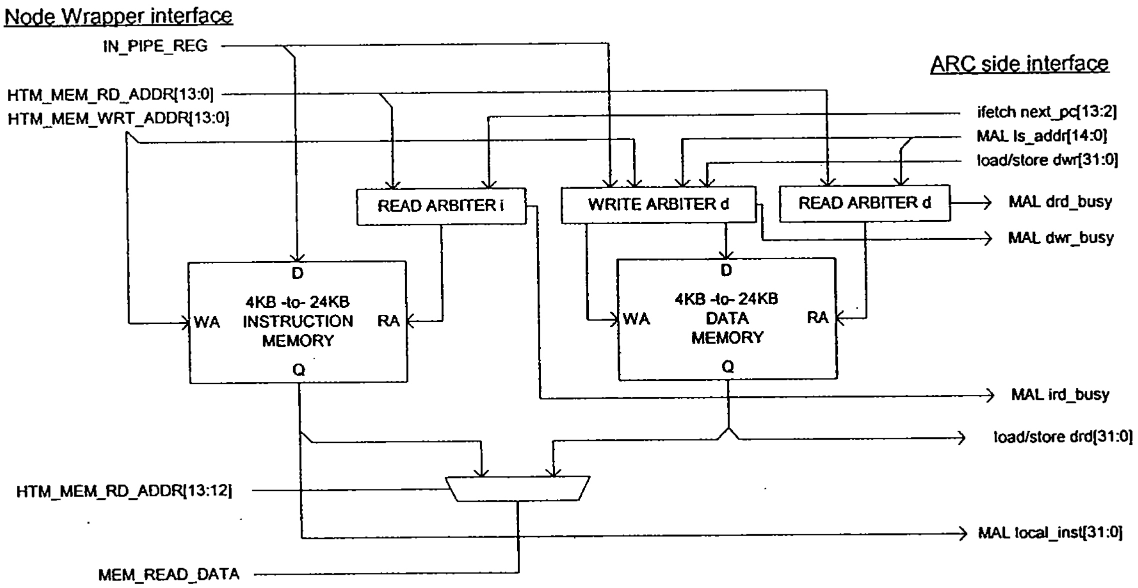
Read Arbiter d

1. Node wrapper read
2. ARC load/store

Write Arbiter d

1. Node wrapper write
2. ARC load/store

Figure 4: Multi-way Memory Arbitration Unit



After the proposal for Nodal Memory Subsystem With Memory Aggregation, the arbitration logics described in this section are pushed into that module.

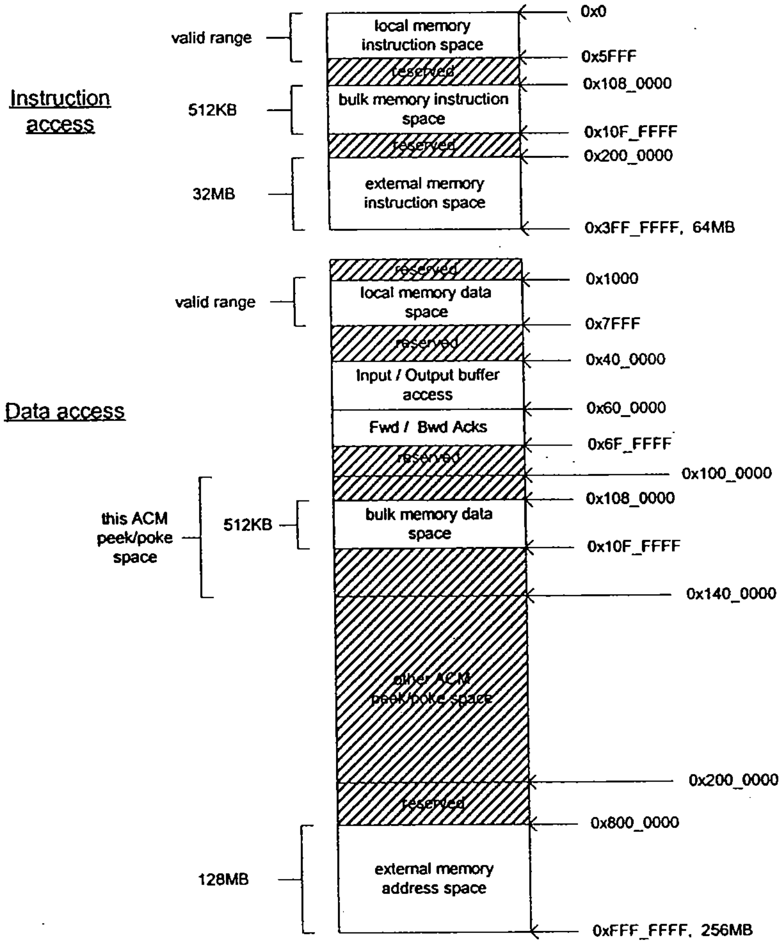
1.2.3 Memory Aperture Logic (MAL)

If transaction were targeted at other nodes, these memory requests would pass directly to the MIN Packet Assembly block, which packetizes the memory request to send to other memory controllers sitting on the MIN. The MAL's main duties are:

- Decodes whether a memory request is 1) local memory access, 2) external memory access (including bulk, external, and peek/poke accesses), or 3) input or output buffer access, from either ifetch or load/store buses.
- Maps, if necessary, load/store memory access addresses to a base physical memory page on the respective memories per task basis (1-to-1 mapping to physical address by default).

Figure 5 tries to portray the memory map as seen by a task running on the PSN. Note, instruction and data accesses are separate and independent. Instruction access actually presents the physical memory address, while the data access per task basis can be translated and mapped unto a physical address different from the module's default point of view.

Figure 5: Module View Default Memory Map



In the MAL block, the configuration registers contain the Memory Access Scope (MAS) parameters for the memories, which, if necessary, could map task's data memory access to actual physical addresses depending the base address set and limited by the configured buffer size (power-of-2). Essentially, this establishes a protected memory page that bounds all load/store requests to a region of the physical memory. Address 0 for the respective memory (local, bulk, or external) on the load/store bus would translate into the physical base address set in the following configuration registers:

Table 9: MAL Configuration Register for Memory Access Scope

Number	Name	Function
0x100	MALB	Memory access local page base address, 4-bit value base address 0000 0x0000 (DEFAULT) 0001 0x0400 0010 0x0800 0011 0x0c00 : : 1111 0x3c00
0x101	MALS	Memory access local page size, 4-bit value size 0000 1KB 0001 2KB 0011 4KB 0111 8KB 1111 all range (DEFAULT)
0x102	MABB	Memory access bulk page base address, 5-bit value base address 00000 0x0000 (DEFAULT) 00001 0x4000 00010 0x8000 00011 0xc000 : : 11111 0x7c000
0x103	MABS	Memory access bulk page size, 5-bit value size 00000 16KB 00001 32KB 00011 64KB 00111 128KB 01111 256KB 11111 all range visible (DEFAULT)
0x104	MAEB	Memory access ext. page base address, 5-bit value base address 00000 0x0000000 (DEFAULT) 00001 0x0200000 00010 0x0400000 00011 0x0600000 : : 11111 0x3e00000
0x105	MAES	Memory access ext. page size, 5-bit value size 00000 2MB 00001 4MB 00011 8MB 00111 16MB 01111 32MB 11111 all range visible (DEFAULT)
0x106 - 0x107	---	reserved
0x108	MASP	Packed MAS parameters (Write only), initializes above parameters in one packed word write (28 bits)
0x109 – 0x11F	---	reserved

If the external memory attached is 64MB (even if the address range allowed is 128MB maximum), the translation of the task's load/store memory access addresses to actual physical memory addresses is calculated as follows:

For local memory:

(load/store address[13:10] & MALS) + MALB => ls_addr[14:10]
ls_addr[14:10] => ls_addr[14:10] for MMAU

For bulk memory:
(load/store address[18:14] & MABS) + MABB => ls_addr[19:14]
ls_addr[18:14] => ls_addr[18:14] for MPA
ls_addr[19] => memory exception, limits ls_addr to all 1's

For external memory
(load/store address[25:21] & MAES) + MAEB => ls_addr[26:21]
ls_addr[25:21] => ls_addr[25:21] for MPA
ls_addr[26] => memory exception, limits ls_addr to all 1's

If a memory access is not within the task's allotted page size, the address wraps to the beginning of the physical memory access range allocated, thus a garbage read or an overwrite situation can occur in this case. In addition, if the memory access goes beyond the physical memory size, i.e. in the case of bulk page base address 0x7c000 with bulk page size 128kB for example, a memory exception will be generated and the address bits held at the last address (at all ones). Furthermore, if the K-node sets the MAS_LOCK register, programs running on the PSN including the SoftKernel can no longer reconfigured these parameters. In other word, the content of the MAS parameter registers cannot be changed unless the MAS_LOCK register content is 0. This feature gives the K-node the ability to secure a memory page for a particular PSN so that all programs running on it can only access a given region of the physical memory.

Table 10: Memory Access Lock Control Register in Node Specific Space

Address	Name	Function
0x1020	MAS_LOCK	Memory access lock, 1-bit, default: 0, not lock

Lastly, high-level block diagrams for the MAL are depicted in Figure 6 and Figure 7.

Figure 6: Memory Aperture Logic load/store request path

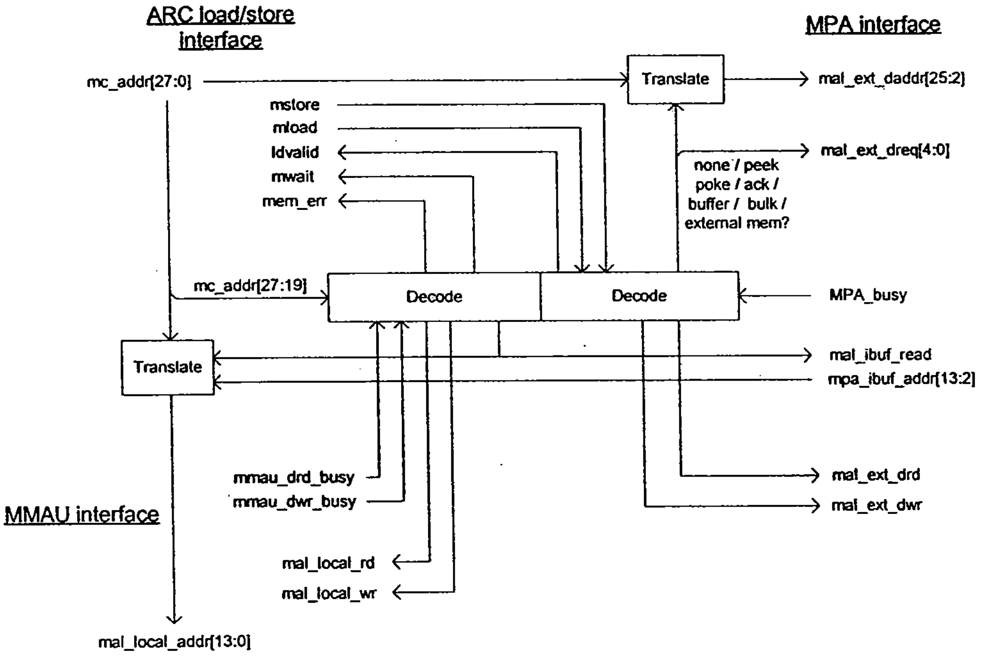
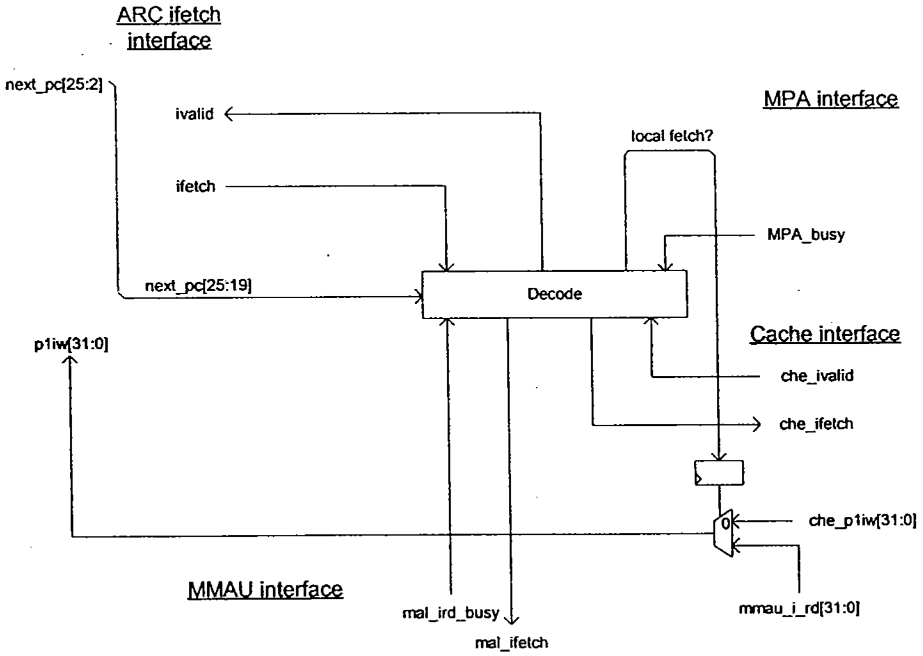


Figure 7: Memory Aperture Logic instruction request path

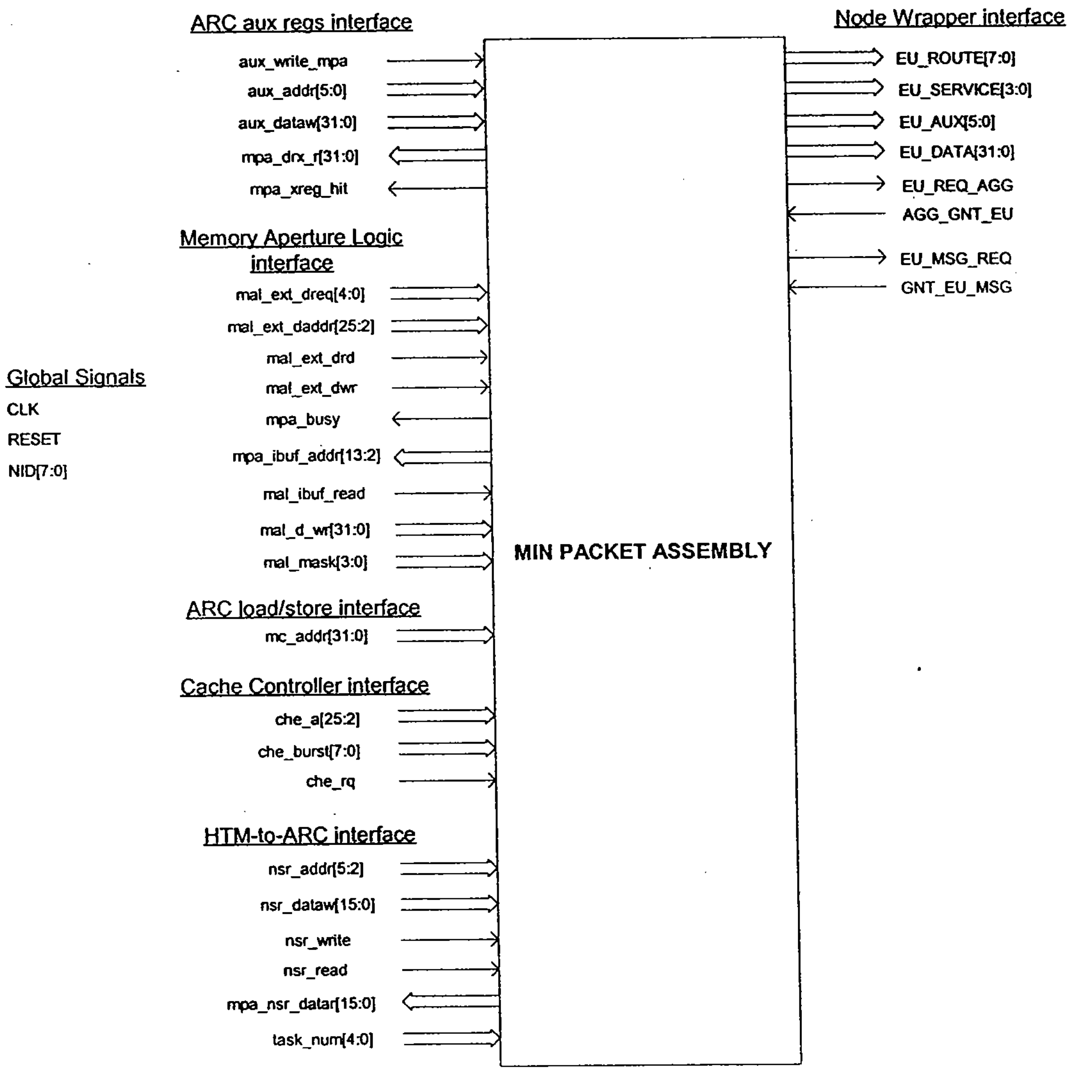


1.2.3.1 Ifetch and Load/store Off Node

A special handling of the memory read request going off-node is built into the PSN. To prevent the case of the system hang while reading (this includes peeking of the K-node) a nonexistent memory address location, a 12-bit time-out timer is instantiated in this block. The timer counts down as soon as an off-node request is generated by the MAL. When the timer wraps after 4096 cycles, a memory exception is generated.

1.2.4 MIN Packet Assembly (MPA)

Figure 8: MIN Packet Assembly Interface Diagram

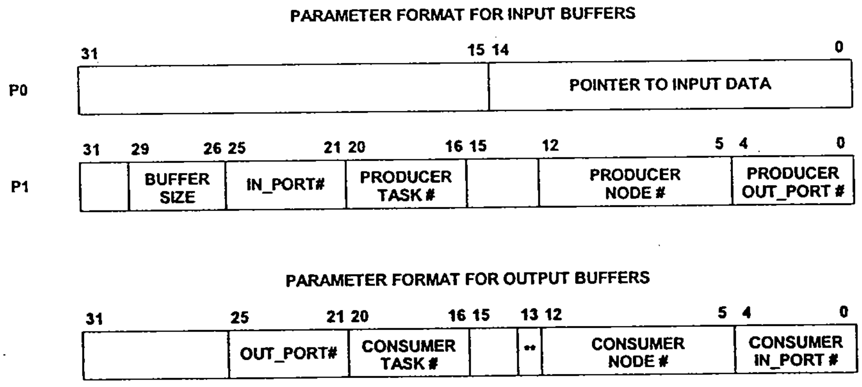


This block takes all MIN traffic passed from MAL and packages them into MIN words for the Data Aggregator in the Node Wrapper. In the MPA block, the configuration registers contain the configuration parameters for each of the input and output buffers obtained from the Task Parameter List (TPL).

Table 11: Buffer Configuration Registers in Auxiliary Space

Number	Name	Function
0x1F0	IBUF_CFG_MODE	Input buffer address pointer update mode, 1-bit 1 = write to 1st input buffer parameter updates only the values in the input buffer current pointer register file; 0 = (default) write to 1st input buffer parameter updates values in both input buffer current pointer and base pointer register file
0x180	INBUF0_P0	1st packed parameter word for input buffer 0 of current task
0x181	INBUF0_P1	2nd packed parameter word for input buffer 0 of current task
:	:	
0x1BE	INBUF31_P0	1st packed parameter word for input buffer 31 of current task
0x1BF	INBUF31_P1	2nd packed parameter word for input buffer 31 of current task
0x1C0	OUTBUF0_P	Packed parameter word for output buffer 0 of current task
0x1C1	OUTBUF1_P	Packed parameter word for output buffer 1 of current task
:	:	
0x1DE	OUTBUF30_P	Packed parameter word for output buffer 30 of current task
0x1DF	OUTBUF31_P	Packed parameter word for output buffer 31 of current task

Figure 9: Packed Buffer Parameter Format

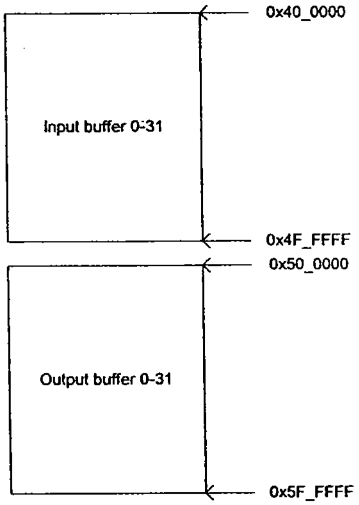


** Consumer Address Mode

1.2.4.1 Input and Output Buffer Access

The input and output buffers are accessed through the memory mapped range as shown in Figure 10.

Figure 10: Input and Output Buffer Access Address Map



In the case of a PTP write, in consideration of stalling the ARC when aggregator doesn't grant, the ST instruction is used to write to a buffer. For example:

```
st  %r0, [%r6]    ; r0 holds the data, and
                  ; r6 holds the proper memory mapped address
                  ; points to the buffer of interest,
                  ; for example, r6 = 0x50_0000 for output buffer 0
```

One can write to any address in the associated buffer access range, i.e. 0x50_0000 ~ 0x5F_FFFF, to initiate a PTP write to an output port. The address bits [7:3] in this range are used to decode which buffer to write (Table 12). The buffer number to port number is translated automatically depending on the initialized buffer configuration parameters. The buffer pointer (offset address within a buffer) should be taken care of by the destination Node Wrapper.

Table 12: Output Buffer Access

Address[7:3]	Hex addr[7:0]	Function
00000	0x0	Indicate which buffer to write to, i.e. buffer 0
00001	0x8	Indicate which buffer to write to, i.e. buffer 1
:	:	
11111	0xF8	Indicate which buffer to write to, i.e. buffer 31
Address[12]	**Hex addr[12:0]**	**Function**
0	0x0---	Regular PTP access
1	0x1---	PTP Packet mode access

Potentially, one could reach maximum one PTP write per cycle throughput in a pipeline, assuming both the aggregator at this node and the distributor at target node kept granting. Of course, there will be some initial overhead needed: one cycle to Output pipeline register + MIN pipeline latency + target Node Wrapper pipeline latencies. At the end of the buffer write sequence, to generate forward ACK to the consumer of the output buffer, the task would have to write to the designated control registers in the MIN Packet Assembly (described in the following MIN Packet Assembly (MPA) section) block to generate this ACK.

Note, the ACM network architecture only supports 32-bit data structure, which means the buffer write should normally done through a ST instruction. In the case of a STW or a STB instruction is performed to the output buffer address, the byte or word data will always be zero-extended to 32-bit to fill up the MIN word data field. The output buffer write will always be done physically to dword address boundary.

To read from the PTP input buffers, a task executing on the ARC uses LD instruction to read to the proper offset address in the buffer. Since the input buffer resides on the local node memory, there can be two ways of reading data from the input buffer: 1) getting the actual physical buffer address from the buffer configuration register (e.g. INBUF0_P0) (Table 11), and loading from the memory directly based on this base address; and 2) reading the input buffer through the memory-mapped address shown in Figure 10. The first method requires the software to track the buffer address pointer and the buffer size boundary that can take up enormous overhead. The second method makes use of an auto-updating data address generator (DAG) in the MPA block, to automatically generate the buffer offset pointer. Table 13 describes how the address in this input buffer access range is decoded. Only address bits [7:2] in the address range 0x40_0000 to 0x4F_FFFF are used to decode 1) which buffer to read and 2) whether to post-increment the DAG.

Table 13: Input Buffer Access

Address[2]	Hex addr[2:0]	Function
0	0x0	Read from buffer without changing the buffer pointer
1	0x4	Read from buffer and post-increment the buffer pointer
Address[7:3]	**Hex addr[7:0]**	**Function**
00000	0x0	Indicate which buffer to read from, i.e. buffer 0
00001	0x8	Indicate which buffer to read from, i.e. buffer 1
:	:	
11111	0xF8	Indicate which buffer to read from, i.e. buffer 31

The amount to post-increment the DAG is determined by the LD instruction: 1) LD increments it by 4; 2) LDW increments it by 2; and 3) LDB increments it by 1. Whenever the buffer pointer is updated, the associate "Pointer to Input Data" value in the buffer configuration register will also be updated.

For example, for reading and shifting 50 32-bit vector elements from input buffer 0:

```
            mov   LP_COUNT, 50
            mov   %r10, 0x400000
            lp    loop_end
loop_in:    ld    %r0, [%r10,0x4]    ;read from buf0 and advance DAG
            asr   %r1, %r0, 3        ;right shift each data by 3
            <save result>
loop_end:
```

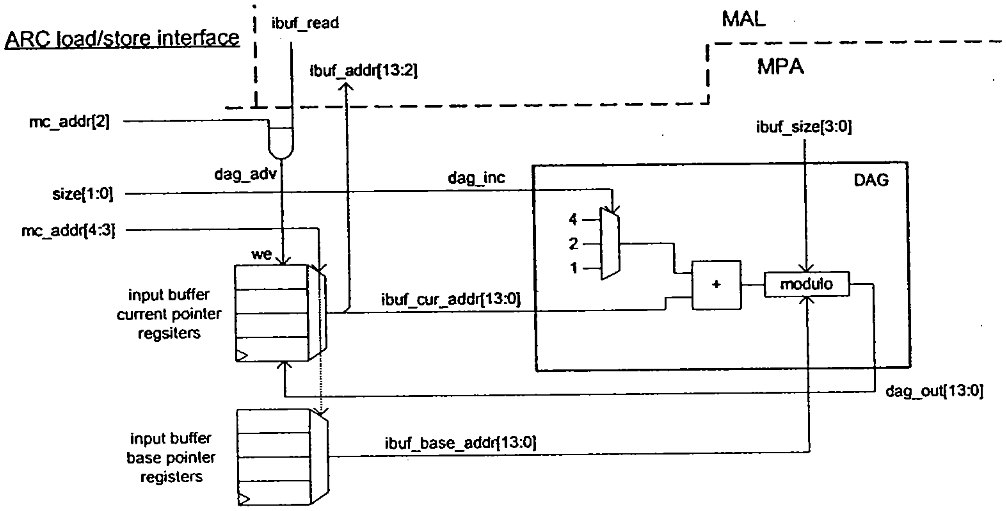
For example, for reading and multiplying 20 16-bit vector elements from input buffer 2 and 3:

```
            mov   LP_COUNT, 20
            mov   %r10, 0x400000
            lp    loop_end
loop_in:    ldw   %r0, [%r10,0x14]  ;read from buf2 and advance DAG
            ldw   %r1, [%r10,0x1c]  ;read from buf3 and advance DAG
            mul64 %r2, %r1, %r0
            <save result>
loop_end:
```

Therefore, one can always retrieve the current buffer data pointer stored from the buffer configuration register. In addition, one can reinitialize the DAG by writing to the INBUFx_P0 register (described in the following MIN Packet Assembly (MPA) section) again with the original input buffer starting address stored in the TPL. Again, this is because the DAG uses the "Pointer to Input Data" value in the buffer configuration register to perform the address generation. Thus, this field for the associated input buffer is updated if requested as the DAG increments. The internal logic of the DAG limits the address from going out of the buffer boundaries.

The block diagram describing the connections between the MAL block and the DAG logics resided in the MPA is shown in Figure 11. The two register sets, one contains the current buffer address pointer and another contains the original buffer pointer base address as in the TPL, are updated when INBUFx_P0 register is written to, depending on the IBUF_CFG_MODE bit. A read to the INBUFx_P0 register reads the value from the current buffer pointer registers set. The design assumes that input buffer boundaries will always be dword aligned.

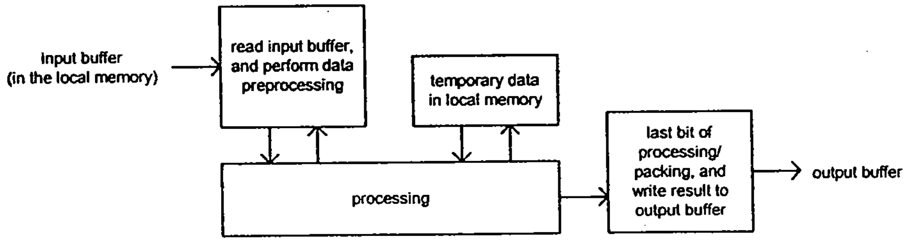
Figure 11: Buffer DAG and Control



At the end of the buffer read sequence, to backward ACK to the producer of the input buffer, the task would have to write to the designated address range described in the following section.

The decision to use ARC's load/store for input buffer read and output buffer write is based on the assumption that a typical task processing flow (Figure 12) and the efficiency of the stalling mechanism mentioned previously.

Figure 12: Typical Task Processing Flow


input buffer (in the local memory)
read input buffer, and perform data preprocessing
temporary data in local memory
processing
last bit of processing/ packing, and write result to output buffer
output buffer

1.2.4.2 Forward and Backward ACKs

To generate forward ACK to the consumer of the output buffer, or backward ACK to the producer of the input buffer, the task program would have to write to the proper memory mapped location as shown in Figure 13, using the ST instruction.

Figure 13: ACK Generation Address Map

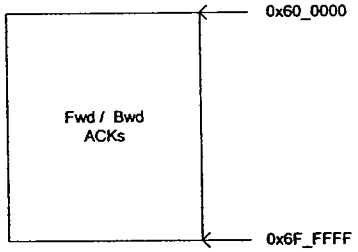
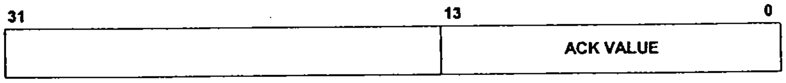


Table 14 describes how the address in this input buffer access range is decoded. Only address bits [9:2] in the address range 0x60_0000 to 0x6F_FFFF are used to decode 1) whether to generate a forward ACK to output buffer or backward ACK to input buffer, and 2) which buffer number for ACK generation. When a store to one of these ACK address occurs, a proper forward/backward acknowledgement will be send depending on the address according to Table 14. The store word should contain the actual acknowledgement value being send located in the longword as described in Figure 14.

Table 14: ACK Generation Address Locations

Address[2]	Hex addr[2:0]	Function
0	0x0	ACK to other node's counter table
1	0x4	ACK to this task's counter table
Address[8:3]	**Hex addr[8:0]**	**Function**
000000	0x00	Indicate which buffer to ack, i.e. buffer 0
000001	0x08	Indicate which buffer to ack, i.e. buffer 1
:	:	
111111	0x1F8	Indicate which buffer to ack, i.e. buffer 63
Address[10:9]	**Hex addr[10:9]**	**Function**
00	0x0–	Select output buffer (Fwd ACK)
01	0x2--	Select input buffer (Bwd ACK)
10	0x4--	Test bit set for Fwd ACK, only if Address[2] = 1
11	0x6–	Test bit set for Bwd ACK, only if Address[2] = 1
Address[12]	**Hex addr[12:0]**	**Function**
0	0x0---	Regular PTP access
1	0x1---	PTP Packet mode access

Figure 14: ACK Generation Store Word Format



1.2.4.3 Point-to-Point Packet Mode Special Case

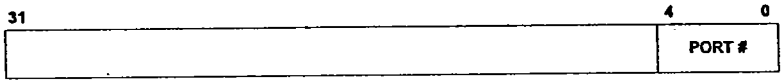
For both output buffer write and ack accesses, in the special case that the packet mode is needed, one sets the address bit [12] when performing those transaction to indicate packet mode, i.e. output buffer write or ACK will be sent via Service Codes 0x2 and 0x3 respectively.

However, before any packet mode transaction is to be performed, one needs to send the packet mode header to the destination node. This can be done by simply write the buffer number that is configured to perform the packet transfer to the auxiliary register PTP_PACKET (Table 15).

Table 15: PTP Packet Mode Header Auxiliary Register

Number	Name	Function
0x1F1	PTP_PACKET	Send Point-to-Point Packet Mode header, write-only

Figure 15: PTP Packet Header Generation Store Word Format



When a write to this register is detected, MPA sends the packet mode header information immediately, pending all other outgoing traffics. The header information is as described in the ACM Nodal Architecture, where the data field looks like:

bits [28:24]	= Source Port Number
bits [20:16]	= Source Task Number
bits [7:0]	= Source Node ID

Be advised that the packet mode transaction should not be used in a SoftGround tasks (described in the following HTM2ARC section) because the MIN path may be locked when a SoftGround task is switch off to background.

1.2.4.4 Full MIN Word Write

To write an arbitrary MIN word to be sent on the MIN for debug purpose, two registers are allocated for this purpose (Table 16). A write (using SR instruction) to MIN_WR_LSP stores the lower 32 bits of the whole 51-bit MIN word structure to a temporary holding register. Only when a write to MIN_WR_MSP with the remaining most significant 19 bits of the MIN word structure (resides in bit 18 down to 0 of the SR instruction) that a MIN packet is actually assembled and sent. When MPA is sending this composed MIN word, all other outgoing traffic will be stalled.

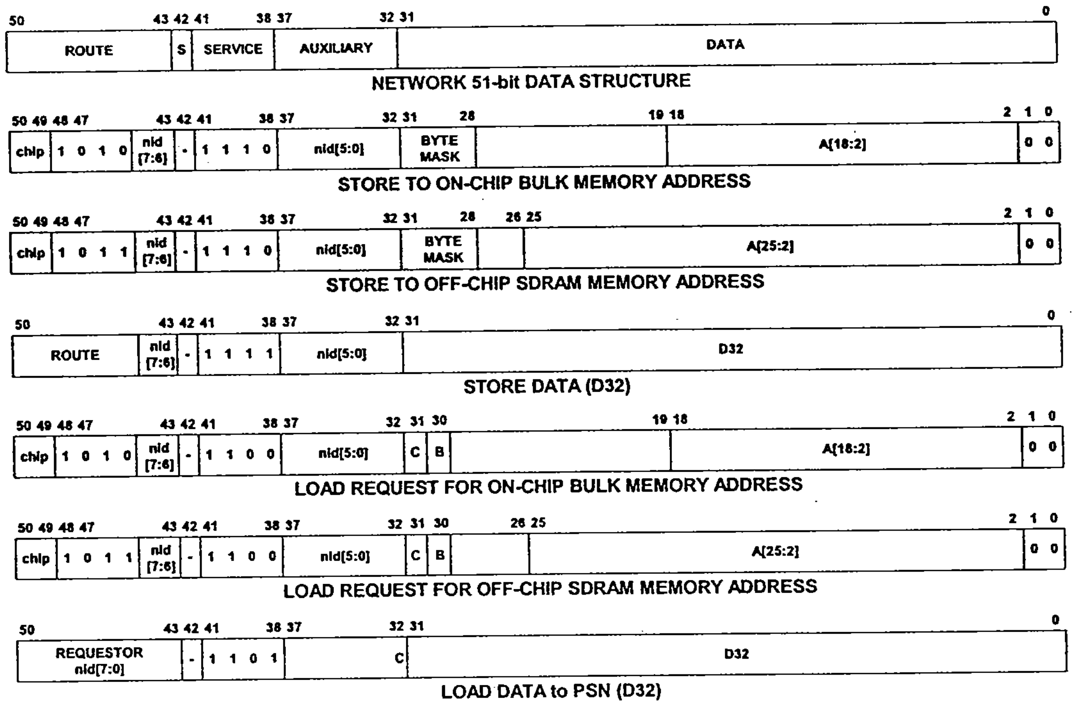
Table 16: MIN Word Write Auxiliary Registers

Number	Name	Function
0x1F2	MIN_WR_LSP	Lower 32-bit of MIN word to send, write only
0x1F3	MIN_WR_MSP	Upper 19-bit of MIN word to send, write only

1.2.4.5 Ifetch and Load/store Off Node

To adhere to the overall MIN data flow model, one should take advantage of the point-to-point facilities in a data processing flow for block data transfer. To prevent backing up the MIN, one should never perform block data pokes or random access writes to the memory controllers, since the target has to consume all incoming traffic. Therefore, the programmer should consult the Memory Controller Specification for the depth of write data queue; and care should be taken so that this write data queue is not overrun. Nevertheless, in the case of random memory access to the memories off-node, the load/store requests will be encoded as in the following to send out to the MIN:

Figure 16: Random Access Request Formats on the MIN



NETWORK 51-bit DATA STRUCTURE

STORE TO ON-CHIP BULK MEMORY ADDRESS

STORE TO OFF-CHIP SDRAM MEMORY ADDRESS

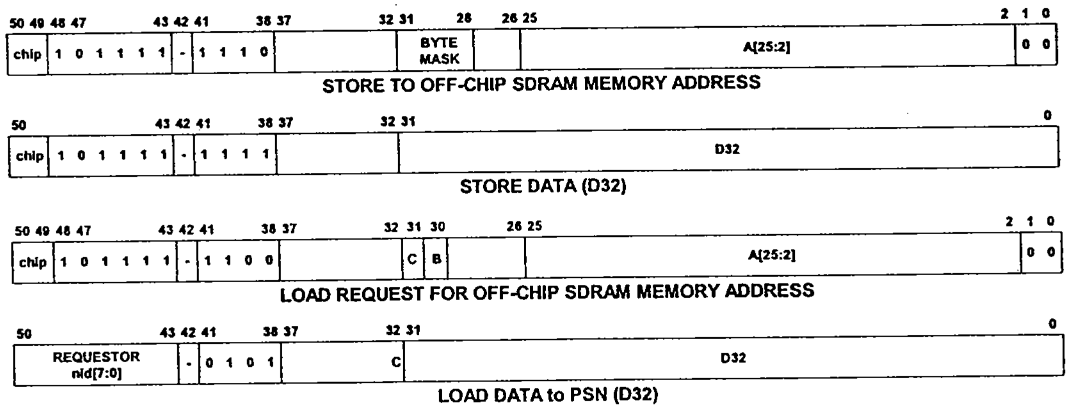
STORE DATA (D32)

LOAD REQUEST FOR ON-CHIP BULK MEMORY ADDRESS

LOAD REQUEST FOR OFF-CHIP SDRAM MEMORY ADDRESS

LOAD DATA to PSN (D32)

Figure 17: Fast-Track Memory Request Formats



STORE TO OFF-CHIP SDRAM MEMORY ADDRESS

STORE DATA (D32)

LOAD REQUEST FOR OFF-CHIP SDRAM MEMORY ADDRESS

LOAD DATA to PSN (D32)

A read request will always return the whole dword on the dword memory address boundary. The sorting of the returned data (for LDB and LDW instructions) is done automatically once the data is received from the network. For a write request, the byte mask bits will be encoded in the most significant two bits of the data field when the write address is sent. The byte or word data will be realigned in the 32-bit data field, so that the byte mask bits represents the locations of the valid bytes to be written.

In the case of a load or a cache line refill request, the same read quest MIN word format would be used. However, the most significant 2-bit in the data field when the load request is sent encodes whether a read request is originating from load or cache and whether a read request requires a 16 consecutive dword return for the cache line according to following table:

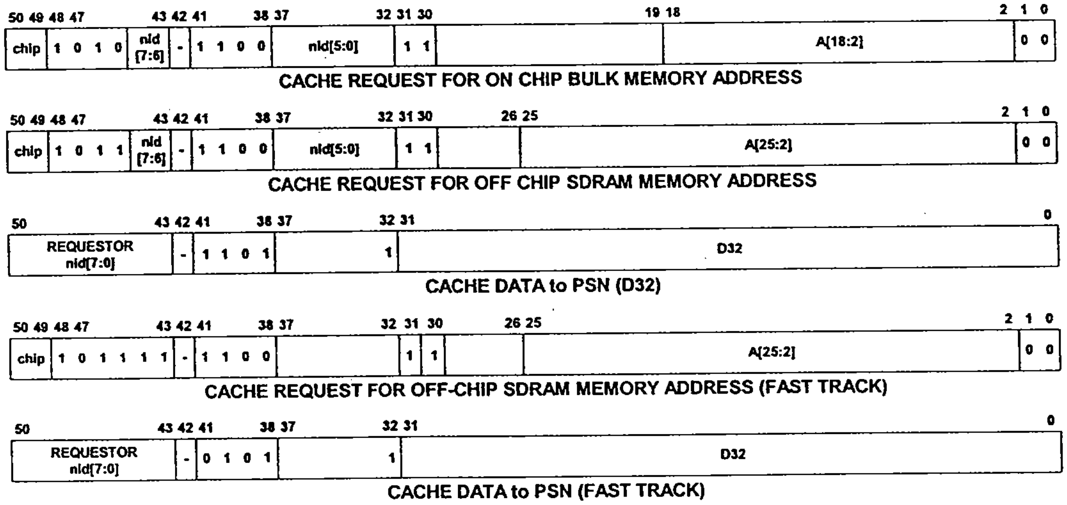
C	B	Description
0	0	Load request, 1 dword
0	1	not used
1	0	Cache request, 1 dword
1	1	Default Cache request, 16 dwords

When the memory controller receive a read request, it responds by sending data back via the memory random access read data MIN service. Since the same MIN word is used for both returning a load or a cache data return, the memory controller should use the LSB of the auxiliary field to encode where a return read data package should route. Thus, bit 31 of the data field in the read request can be passed back to indicate a load or a cache return, for example:

Auxiliary [0]	Description
0	Read data to Load/Store Unit
1	Read data to Cache Controller

Note, a cache request will normally ask for a full cache line return of a 16-dword burst, if the cache is not disabled. In the case of a cache miss, the starting address of the cache line will be supplied to the memory controllers along with both C and B bits set to one in the following format. In return, 16 consecutive read data should return as described in the follow:

Figure 18: Default Cache Request Format on the MIN



CACHE REQUEST FOR ON CHIP BULK MEMORY ADDRESS

CACHE REQUEST FOR OFF CHIP SDRAM MEMORY ADDRESS

CACHE DATA to PSN (D32)

CACHE REQUEST FOR OFF-CHIP SDRAM MEMORY ADDRESS (FAST TRACK)

CACHE DATA to PSN (FAST TRACK)

1.2.5 HTM-to-ARC

Figure 19: HTM-to-ARC Interface Diagram

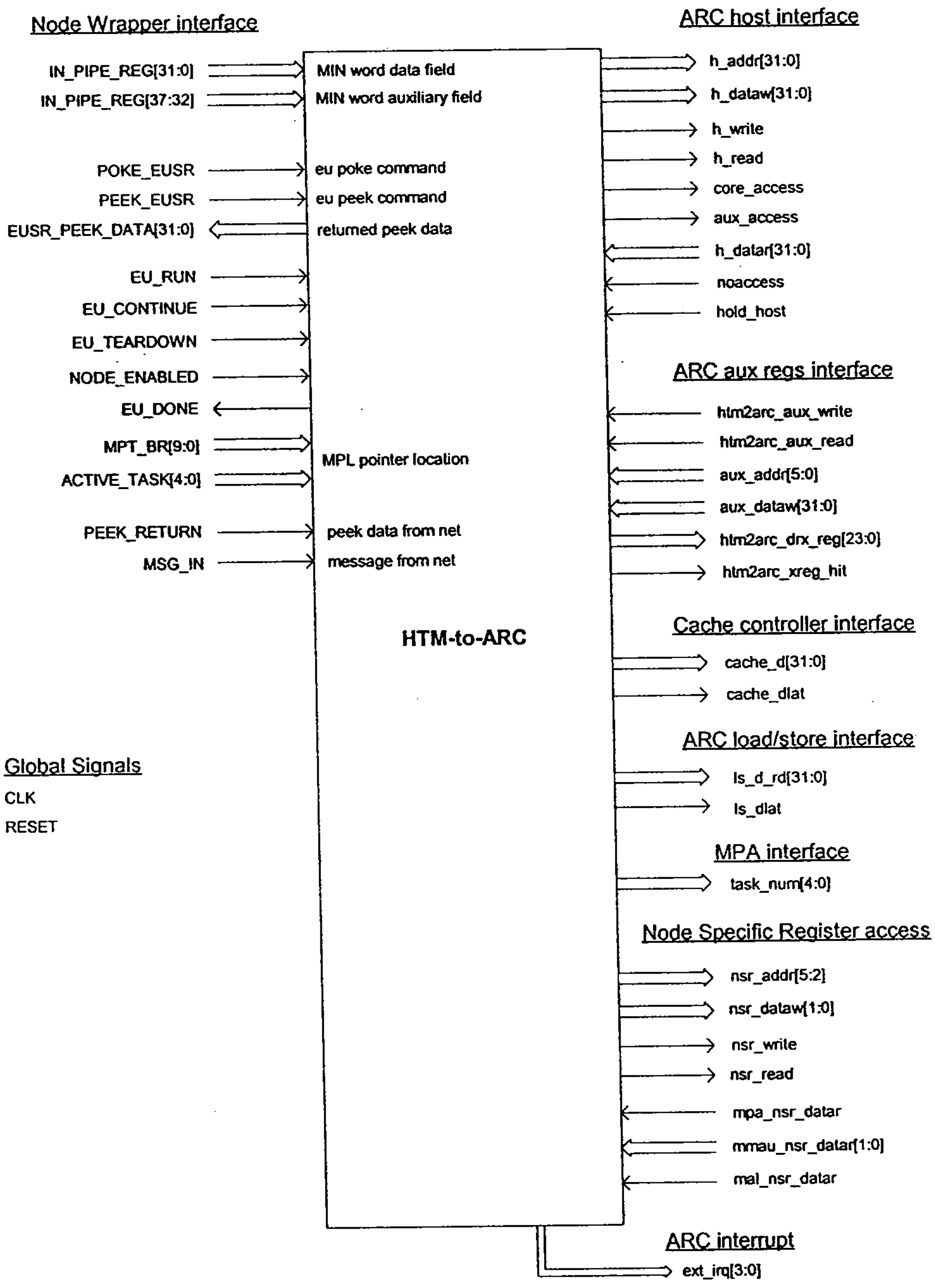


Figure 19 shows the interface block diagram for HTM-to-ARC.

HTM-TO-ARC block mainly serves as an adapter for HTM Peek/Poke Module, HTM FSM, and TPL Pointer Table interfaces to the host interface of the ARC. PSN and K-node use the same standard Node Wrapper with some additions discussed in the following Assumptions and Issues section. The ARC host access would be mapped to the 13-bit Node-Specific Registers field described in the figure: Memory Map for Tables and Register in the Hardware Task Manager document.

Table 17: Node I/O Address Map to ARC Host Access

Address Range	
0x0000 – 0x07FF	ARC core register access
0x0800 – 0x0FFF	ARC auxiliary register access
0x1000 – 0x1FFF	Node specific access only registers

The address in Table 17 is encoded in the lower order bits (A[13:0]) of the data field in the "POKE NODE I/O ADDRESS" or "PEEK NODE I/O ADDRESS" MIN words. The core and auxiliary register numbers supplied throughout this document translate to dword-granularity addresses, meaning they are dword-aligned when address-mapped. For example, core register 0x1 is mapped to address 0x4, and auxiliary register 0x2 is mapped to 0x808, and so on, in the increment of 0x4.

K-node through MIN can only access the part of the ARC registers, depending on whether ARC is running or halted as shown in. While the ARC is running, the Node I/O will not have access either to the main core registers or to the auxiliary registers. The exceptions to this are some auxiliary registers: status (aux. reg. 0x0), semaphore (aux. reg. 0x1), and identity (aux. reg. 0x4). The node specific access registers such as SCR, MACR, MAS_LOCK, and MSG_MB will always be accessible through the MIN and take priority over ARC access, meaning at the rare case that the node specific register is being modified by a poke and the ARC is also writing to the same register, the ARC write is ignored. To read or modify other registers, the K-node needs to halt the ARC, by writing 0x200000 to the status register, and perform the necessary peek or poke to the core or auxiliary registers. Otherwise, if poking while ARC is running, the poke command is ignored, leaving the target unchanged; and, if peeking while ARC is running, a null data (0x0) will be returned as the peek data.

Table 18: MIN Accesses to ARC

	ARC Running	ARC Halted
Auxiliary Registers	Mainly No access	Read/Write
Core Registers	No access	Read/Write

1.2.5.1 Port Counter Sign Bit Access

Each port's counter sign bit can be read through auxiliary register range 0x140 ~ 0x17F. Bit [5:0] of the address range is passed onto the Node Wrapper, which identified the input or output and its port number interested.

Table 19: Port Counter Sign Access in Auxiliary Space

Number	Name	Function
0x140	IPORT0_SIGN	Port counter sign of input port 0
:	:	:
0x15F	IPORT31_SIGN	Port counter sign of input port 31
0x160	OPORT0_SIGN	Port counter sign of output port 0
:	:	:
0x17F	OPORT31_SIGN	Port counter sign of output port 31

1.2.5.2 Task Setup

When a task is set to RUN by the HTM, logics in HTM-to-ARC stores the TPL pointer table entry number and the active task number into the TPLP auxiliary register, follows by asserting an interrupt vector #6 (level 2). A vector table for interrupts other than the reset vector resides at the very beginning of the instruction memory (starting from address 0x0) would indicate a jump to the iTaskWrapper routine somewhere in the instruction memory. A sample vector table is described as follows:

```
mem_err:    jal iExceptHandle      ; interrupt vector for memory error
ins_err:    jal iExceptHandle      ; interrupt vector for instruction error
ivector3:   jal timer0_isr         ; jump point for level 1 priority, vector 3
   :            :
ivector6:   jal iTaskWrapper       ; jump point for level 2 priority, vector 6
   :            :
ivector8:   jal timer1_isr         ; jump point for level 2 priority, vector 8
```

The Task Setup part of the iTaskWrapper interrupt service routine (ISR) for example needs to execute the following procedures:

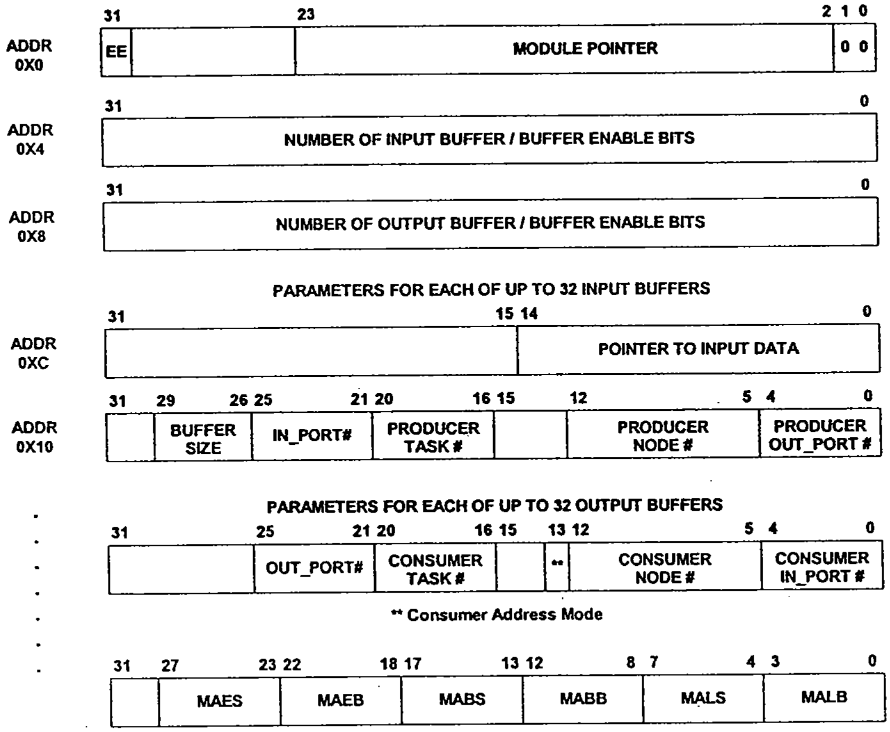
1. Enable interrupt level 2 so that watchdog timer interrupt isn't missed while a task executes, since another level 2 interrupt will still be serviced;
2. Obtain starting address of TPL via, LD %r0, [TPLP], where %r0 gets the TPL starting address;
3. Load Task Configuration parameter (Module pointer, Execution Environment, Number of Buffers in packed format, the very first parameter) and store it to TASK_CFG register;
4. Retrieve Execution Environment (EE) parameter, and if EE = 1, indicating a SoftGround task, then go to the SoftGround task registration sequence described below, otherwise continue the following;
5. Retrieve Buffer Enables parameters and store them to IBUF_EN and OBUF_EN registers, use these numbers for loop setup of following step as well;
6. Load configuration parameters for each of the input/output buffers (packed format, Figure 20) from the memory and store them to configuration registers in MPA;
7. Load Memory Access Scope parameters (packed format) from the memory and store them to MASP configuration register in MAL, if needed; and
8. Load Module Pointer (Task's starting PC) from MOD_PTR register, and execute the module via JL Module Pointer.

Table 20: Some Task Configuration Registers in Auxiliary Space

Number	Name	Function
0x127	TPLP	TPL pointer table entry number, 15-bit, word address [15:1], bit [5:1] represents task number
0x128	TASK_CFG	Write only, Task Execution Env, Module pointer
0x129	TEE	Read only, task execution environment, 1-bit, 0=RealGround task, 1=SoftGround task
0x1F4	IBUF_EN	Number of input buffers / buffer enable bits for this task, 32-bit, e.g. 'h00000007 enables 3 input buffers: 0, 1, and 2.
0x1F5	OBUF_EN	Number of output buffers / buffer enable bits for this task, 32-bit, e.g. 'hffffffff enables all 32 output buffers.
0x12A	MOD_PTR	Read only, module pointer, 22-bit, dword address [23:2]

The Task Configuration Register (TASK_CFG), written to as one dword packed configurations, which includes EE (execution environment) and Module Pointer fields, two fields that can be read back independently via TEE and MOD_PTR registers respectively. The suggested TPL format in the memory should be the following:

Figure 20: Task Parameter List Format in Memory



1.2.5.3 Ack, Test & Wait Stage, and Task Teardown

When task runs to completion, it is the task module's responsibility to perform all ACKs required and the final ACK, which sets the test_after_ack bit. Then the module program returns to the iTaskWrapper service routine via JAL BLINK; and the Teardown part of iTaskWrapper ISR goes thru the following sequence for example:

1. Reset MASP configuration register in MAL to default value, if needed;
2. Update the Physical Memory Address fields in the input buffer configuration parameters in the TPL with the updated values in the INBUFx_P0 registers;
3. Poll TASK_CONTINUE register, if set, jump to the task's starting PC to rerun the task without the setup sequence;
4. Poll TASK_TEARDOWN register, if not set, repeat steps #3 and #4 again (this is the TEST_and_WAIT state); and
5. Produce the EU_DONE signal for the HTM FSM by writing to TASK_DONE auxiliary register, and jump to the SoftGround dispatcher routine that runs other soft scheduled tasks if they are registered in the SoftKernel or go into the background sleep-and-wait routine (BackGround task) waiting for future HTM task run entry interrupt.

EU_CONTINUE and EU_TEARDOWN signals are registered and mapped to two auxiliary registers in the ARC, TASK_CONTINUE and TASK_TEARDOWN read-only auxiliary registers respectively. The EU_DONE will also have a software pointer in the TASK_DONE write-only register.

Table 21: Some Task Control/Status Registers in Auxiliary Space

Number	Name	Function

Number	Name	Function
0x120	TASK_CONTINUE	Read only, latches by EU_CONTINUE signal from the node wrapper, and cleared when read
0x121	TASK_TEARDOWN	Read only, latches by EU_TEARDOWN signal from the node wrapper, and cleared when read
0x122	TASK_DONE	Write only, generates the EU_DONE signal for the HTM
0x1F6	TASK_GO	Write only, modifies this task's GO bit in the HTM State Information Table, 1-bit

1.2.5.4 Task Management via SoftKernel

When a task is designated to be scheduled by a SoftKernel, the EE parameter in the TPL should be set to one at initialization. This is detected in the iTaskWrapper ISR and instead the following SoftGround task registration sequence is executed for example:

1. Disable the task in the HTM by writing 0 to the GO-bit of the task's entry in the State Information Table (This is done by SR 0, [TASK_GO]);
2. Add the task to the SoftGround queue (including saving the content of TPLP), and do not execute until there's no task ready to run in the HTM task queue;
3. Load configuration parameters for only ONE buffer (packed format) from the TPL memory and store them to buffer configuration registers in MPA;
4. Perform this task's ACK to this buffer with the test_after_ack bit set but use an ACK value of 0; and
5. Begin polling the TASK_TEARDOWN register, set TASK_DONE when TASK_TEARDOWN is set, and execute to JAL [ILINK2] to jump back the SoftGround dispatcher routine to run queued tasks.

If no task were ready to run in the HTM queue, a SoftGround dispatcher could execute tasks already registered in the SoftGround queue. Similar to the RealGround task setup steps, the SoftGround dispatcher routine would look for example:

1. If there is no task in the SoftGround queue, then sleep, otherwise continue;
2. Restore TPLP register content;
3. Obtain starting address of TPL via, LD %r0, [TPLP], where %r0 is the TPL starting address;
4. Load Task Configuration parameter and store it to TASK_CFG register;
5. Retrieve Buffer Enables parameters and store them to IBUF_EN and OBUF_EN registers, use these numbers for loop setup of following step as well;
6. Load configuration parameters for each of the input/output buffers (packed format) from the memory and store them to buffer configuration registers in MPA;
7. Load Memory Access Scope parameters (packed format) from the memory and store them to MASP configuration register in MAL, if needed; and
8. Load Module Pointer (Task's starting PC), and execute the module via JL Module Pointer.
9. Module executes and returns via JAL BLINK.
10. Reset MASP configuration register in MAL to default value, if needed;
11. Perform ACKs required based upon the Number of Buffers parameters; in these ACK's, the test_after_ack bit is never set;
12. Re-enable the task in the HTM by writing 1 to the GO-bit of the task's entry in the State Information Table (This is done by SR 1, [TASK_GO]); and
13. Go to step 1.

1.2.5.5 iTaskWrapper ISR

Under the above proposal, a RealGround task (task registered only on the HTM) will always have higher priority for execution because of its level-2 interrupt entrance. Therefore, a SoftGround task could be interrupted by any new task queued to run at the HTM. At the onset of a EU_RUN induced interrupt and at the entrance to the iTaskWrapper ISR, one needs to first store the full context frame for a possible original running SoftGround task. This means saving the content of registers indicated in Table 4 and Table 5, plus the following:

1. Reset MASP configuration register in MAL to default value, if needed;
2. Retrieve only INPUT Buffer Enables parameter, for loop setup of next step;
3. Store current input buffer pointers (read from INBUFx_P0 registers).

At the exit of the ISR, all abovementioned data need to be restored to proper location. To restore, the following needs to be performed: the steps 2-7 described in the above SoftGround dispatcher routine needs to be performed again. However, before

1. Restore all contents of registers mentioned in Table 4 and Table 5;
2. Repeat steps 3-7 described in the above SoftGround dispatcher routine;
3. Retrieve only INPUT Buffer Enables parameter, for loop setup of next step;
4. Restore current input buffer pointers by writing to the necessary INBUFx_P0 registers with the IBUF_CFG_MODE register set; this ensures only the register file that keeps the current input buffer addresses is updated but not the base input buffer address register file.

1.2.6 Timer System

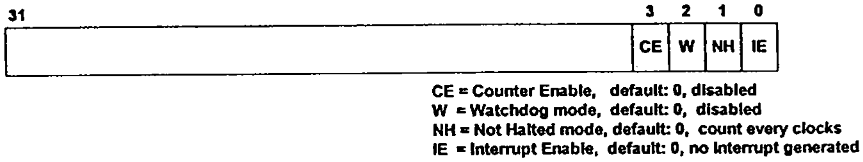
The two ARC 32-bit timers will be used. The control registers for these timers are extended to include an enable bit, which default to disabled. According to ARC documentation, the control register allows one to count all cycle or count cycles when the processor is not halted and to trigger an interrupt when the preprogrammed limit is reached. The Timer starts counting from the "count" value upwards till it reaches the "limit" value after which a level interrupt is generated if it's enabled. It then automatically restarts to count from 0 upward until it reaches the limit value again and the cycle is repeated until the control register is updated. It is up to the software to clear the timer interrupts. Once an interrupt is generated, it is cleared by the action of writing to the "control" register during the interrupt service routine. Timer 1 triggers interrupt vector 8 at level 2 priority, and Timer 0 triggers interrupt vector 3 at level 1 priority.

Table 22: Timer Auxiliary Registers

Number	Name	Function
0x21	T0_COUNT	Timer0 count value, 32-bit, 0x0 on reset
0x22	T0_CONTROL	Timer0 control register, 0x0 on reset
0x23	T0_LIMIT	Timer0 counting limit value, default: 0x00FFFFFF
0x61	T1_COUNT	Timer1 count value, 32-bit, 0x0 on reset
0x62	T1_CONTROL	Timer1 control register, 0x0 on reset
0x63	T1_LIMIT	Timer1 counting limit value, default: 0x00FFFFFF

The timer Limit register is set to 0x00FFFFFF at reset for full backward compatibility. If updated then it can always be reset to 0x00FFFFFF for backward compatible mode of operation. The "count" register may be updated even when the timer is running in which case the internal count register is updated with the new count value and the timer starts counting up from the updated value.

Figure 21: Timer Control Register Format


31
3
2
1
0
CE
W
NH
IE
CE = Counter Enable, default: 0, disabled
W = Watchdog mode, default: 0, disabled
NH = Not Halted mode, default: 0, count every clocks
IE = Interrupt Enable, default: 0, no interrupt generated

To generate a watchdog-reset signal is similar except that (W) control bit should be set. Note that it takes two cycles after reaching the limit value when the system reset signal will be activated (it takes one cycle for the interrupt to be activiated). If both the (IE) and (W) bits are set then only the watchdog reset is activiated since the reset will clear the ARC and the interrupt will be lost anyway.

1.2.7 Interrupt System

Same ARC 2-level priority mask-able interrupts are used. The number is extended to total of 9 interrupts: 3 internal (exceptions: reset, mem_err, and ins_err), 2 for timers (one at level 1 and another at level 2), 1 additional level 1 driving from HTM-to-ARC, and 3 additional level 2 driving from HTM-to-ARC.

Table 23: Interrupt Table

Irq vector	Priority level	Function
0	-	Reset vector
1	-	Memory error
2	-	Instruction error
3	1	Timer 0 interrupt
4	1	Incoming message
5	2	Task abort
6	2	Task run, EU_RUN trigger
7	2	Message buffer full
8	2	Timer 1 interrupt

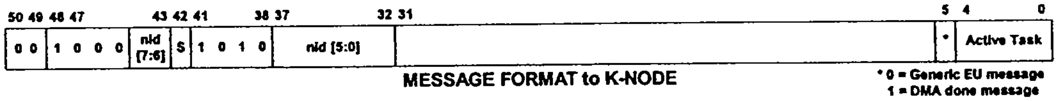
The interrupt system expects a vector table residing at instruction address 0x0 for all interrupts except the reset vector. The reset vector will be hardcoded to address 0x0 of the internal boot ROM in the K-node (address 0x8000). In the vector table, two dwords are needed for the jump point instruction into ISR for each interrupt vector. An example of the vector table can be found in the previous Task Setup section in the HTM-to-ARC description.

For exception interrupts on memory and instruction errors and for task abort interrupt, the interrupt jump vector should point to a routine that executes either the teardown section of a RealGround task or the teardown section of a SoftGround task depending upon what kind of task is current running, which can be identified by the TEE auxiliary register.

1.2.8 Messaging System

The following figure shows a message format on the MIN.

Figure 22: Message Format to K-node



PSN sends a message request by writing a 3-bit message, representing up to 8 different agreed upon messages with the K-node, to the MSG_SND auxiliary register. The 3-bit message can be retrieve through the MSG_MB node-specific register by the K-node. The MSG_SND auxiliary register read and write format are different. If the K-node has retrieved the message, MSG_SND register returns a zero; if not, it returns a one.

By writing to MSG_SND, the EU_MSG_REQ signal is asserted and remains asserted until the grant (GNT_EU_MSG) is received from the node wrapper. In the meantime, the PSN can continue on with processing while the message is being passed and serviced. When the GNT_EU_MSG is asserted (for one cycle only), the EU_MSG_REQ would be de-asserted by the next active clock edge.

Table 24: Messaging Auxiliary Register

Number	Name	Function
0x1F7	MSG_SND	Write to send a message to the K-node, 3-bit message code stored in MSB_MB. Read to confirm retrieval of the message (0=retrieved, 1=not yet retrieved).

Table 25: Message Mailbox Node Specific Register

Number	Name	Function
0x1008	MSG_MB	Message mailbox for K-node to peek, 3-bit message.

1.3 Performance Description

In Northstar, the ARC nodes should be designed to meet the system clock of the 200 MHz. ARC3 core runs at approximately 1 MIPS/MHz, according ARC Concise Guide. In general, each ARC instruction takes only single cycle to execute with the exception that multiply instructions are multi-cycle instructions, where the result will be available after 4 cycles as estimated by ARC. However, this count could vary depending on the technology cell and targeted ARC clock speed. For example, from preliminary synthesis result using TSMC 0.13um general process conservative timing and Synopsys DesignWare parts, the path delay through a 32x32 multiplier is about 8 ns, meeting the max delay constraint of 10 ns. For the target clock speed of 200 MHz, the multiplication would take 2 cycles to complete. Furthermore, as long as the instructions following a multiply is not using the result from the multiplier, ARC pipeline will not be stalled.

Table 26: Performance Matrix at 200 MHz

operation	MOPS	comments	cpi
MUL64 (32x32)	100	assume reg-to-reg operation	2
other arithmetics	200	assume reg-to-reg operation	1
	Mega access / sec		access / cycle
Memory Access	200	assume local memory accesses to/from registers	1

The most likely worst timing path that would limit the operation clock speed of the ARC nodes is the ifetch request to the local memories. This path starts from the ARC pipeline controller, which generates the next_pc address for the following instruction, and ends at the physical nodal memories, passing through arbitration checks and local memory aggregation logics. Similarly, another likely candidate is the local load/store access to the nodal memory path, in which logics are inserted to allow local memory page configuration. Depending where the physical nodal memory is located, these paths may need to modified or simplified to meet the timing target of 200 MHz access.

1.3.1 Simple Load/Store Timing

ARC is designed with a four-stage pipeline and requires that all the instructions in the pipeline are actually executing at different stages in the process to avoid stalling or flushing of the pipeline.

^{1} Pipeline Stage 1 – Instruction fetch

^{2} Pipeline Stage 2 – Operand fetch

^{3} Pipeline Stage 3 – ALU operations

^{4} Pipeline Stage 4 – Write back to core/internal registers

Based on the descriptions of the ARC instruction pipeline, assuming the best-case timing scenario where instructions are all in the cache or the local memory executing in a zero-overhead loop and the memory read access has zero latency, a typical memory transfer sequence would be depicted as in Table 7.

Table 27: Memory Transfer Timing (Best Case)

	t+0	t+1	t+2	t+3	t+4	t+5	t+6	t+7	t+8
ld.a r10, [r11,4]	ifetch^1	r11^2	calc^3	wb r11^4	wb r10^4				
st.a r10, [r12,4]		ifetch^1	r10,r12^2	*stalled*	calc^3	wb r12^4			
ld.a r10, [r11,4]			ifetch^1	*stalled*	r11^2	calc^3	wb r11^4	wb r10^4	
st.a r10, [r12,4]					ifetch^1	r10,r12^2	*stalled*	calc^3	wb r12^4
ld.a r10, [r11,4]						ifetch^1	*stalled*	r11^2	calc^3
st.a r10, [r12,4]								ifetch^1	r10,r12^2

Approximate transfer rate can be expressed as: 3(n-1)+6 where n represents the number of dwords to transfer. For example, for a 1kB transfer, it would take 771 processor cycles to complete.

See ARC Programmers Reference Manual for further details on timing descriptions of different instruction scenarios.

1.3.2 Cache Timing

Only attempts to fetch instruction from memories other than the local node memory will the ifetch request passed unto the i-cache controller. Upon a cache miss, an entire cache line is fetched from the memory sources. The refill of cache line that contained the missing instruction always starts from the beginning of the line. The Cache Bypass mechanism in the ARC will be used to minimize cache miss penalty, where the processor is stalled only until the requested instruction within the cache line has been fetched. The missing instruction is passed to the processor and the processor subsequently is restarted, while the rest of the cache line refill continues. Therefore, the cache performance can be described:

min stall time = 2 + w, and

max stall time = 2 + 16 + w,

where w = wait states for memory read return, and cache line length = 16. Therefore, on average, the cache miss penalty will be (2 + 8 + w). Note, for example, the wait state for a cache refill return from the memory controller could include about 2-cycle round trip on the MIN through "fast track" connection from the PSN to the memory controller, plus any additional cycles needed by the memory controller to fetch the data.

For optimal performance, care should be taken to put any loops in the program code to be within 16 instructions and starts on 64 byte (cache line size) boundaries.

1.4 Node Size Estimate

The following table estimates the PSN die size. Note that this estimate does not include the area required by the Node Wrapper elements and the node memory.

Table 28: Node Size Estimate in 0.13um

Basic Data from Artisan Library for TSMC 0.13um generic process				
Element	Hi	Wide	Sq um	Comments
2 input NAND	3.7	1.4	5.1	Artisan NAND2X1
Reg FF	3.7	9.7	35.6	Artisan DFFRHQX1
Multiplexer, 4 inputs: per input	3.7	2.2	8.1	Artisan MX4X1
Adder (full, bit slice)	3.7	14.7	54.3	Artisan ADDFHX2
SRAM cell, 1 KB (256x32)	125	405	50737	Artisan SRAM-SP-HS
Regfile cell, 2-port, 32x32	108	133	14361	Artisan HS-RF-2P
Regfile cell, 1-port, 16x16	84	75	6310	Artisan HS-RF-1P
Design Elements				
Element	Size	Routing	Sq um	Comments
Gate, in layout @ 70% wiring	5.1	1.43	7.3	Artisan 2-input NAND
Multiplexer	8.1	1.43	11.5	Artisan multiplexer
Reg FF	35.6	1.60	57.0	Artisan DFF
Node Size Estimate				
Element	Gate	Size	Sq um	Comments
ARC 3 basecase core	8,400	7.3	61,106	using Gate
ARC standard extensions:				
32x32 barrel shifter	2,100	7.3	15,277	using Gate
32x32 multiplier	12,900	7.3	93,842	
swap	200	7.3	1,455	
min/max	500	7.3	3,637	
normalize	400	7.3	2,910	
I-Cache logic	1,600	7.3	11,639	using Gate
Memory Arbitration Unit	2,000	7.3	14,549	
Memory Aperture Logic	3,000	7.3	21,824	
Node wrapper adapter logic:				
MIN packet assembly	3,500	7.3	25,461	using Gate
HTM-to-ARC	3,000	7.3	21,824	
Total logic size	**37,600**		**273,524**	
Element	Size		Sq um	Comments
Register File	1,024		28,721	Artisan RF-2P
I-Cache data (1 KB)	8,192		50,737	Artisan SRAM-SP
I-Cache tag	240		6,310	Artisan RF-1P
Total memory size			**85,768**	
Size @ 0.13, sq um			359,292	
Size @ 0.13, sq mm			0.359	
Layout Margin			10%	
Size sq mm @ 0.13um, sq mm			**0.395**	

The size estimate in reality can be much lower depending on the synthesis and layout process. For example, a preliminary synthesis of the 32x32 multiplier yields an area of about 50,000 sq um, using TSMC 0.13um general process standard cell and Synopsys DesignWare library while meeting a max delay path constraint of 10 ns.

1.5 Power Estimate

According ARC3 datasheet, a basic ARC core without a cache would consume around 0.5mW/MHz on a generic 2.5v 0.25um process. Scaling down to 1.2v 0.13um process, the power consumption would be approximately 0.11mw/MHz. Factoring in the inclusion of cache, standard extensions, and other logics, which increases total PSN core logic size to about 4.5 times of a basic ARC3 core, the power consumption for the proposed Northstar PSN core logic would be roughly 0.49mW/MHz. Since ARC performs approximately 1 MIPS/MHz, Table 8 shows theoretical power consumption for the PSN processing core at maximum processing rate.

Table 29: Power Estimate at Maximum Throughput

Feature	Value	Units	Comments
Logic power/MHz	0.49	mW	at 0.13um
Memory power	4.60	mW	full access, equal r/w, at 100 MHz
Max input throughput	400	MB/sec	32 bit data for 100 MHz MIN
Power	53.6	mW	at 100 MHz
Power/Mbyte/sec	0.134	mW	at max data throughput

Low Power features

The actual average power consumption could be lower, since this is a conservative estimate at theoretical maximum processing rate. The theoretical maximum processing rate is rarely present in a typical application run. In addition, ARC's low power features are incorporated, for example, the clock gating option. The clock tree driving ARC's pipeline is gated automatically whenever it is halted or in sleep, while still allowing host access, meaning HTM / MIN access, to touch all configuration status registers and memories. To be exact, the clock is gated only if all of following conditions are met:

1. The ARC is halted (en = '0') or is sleeping after the pipeline has been flushed (sleeping = '1' and AP_p3disable_r = '1').
2. Memory request is not being serviced (lpending = '0', mem_access = '0')
3. The host is not accessing the ARC (host_rw = '0)
4. There are no interrupt requests to service (p123int = '0')
5. The ARC is not in production test mode (test_mode = '0'), such as during scan tests.

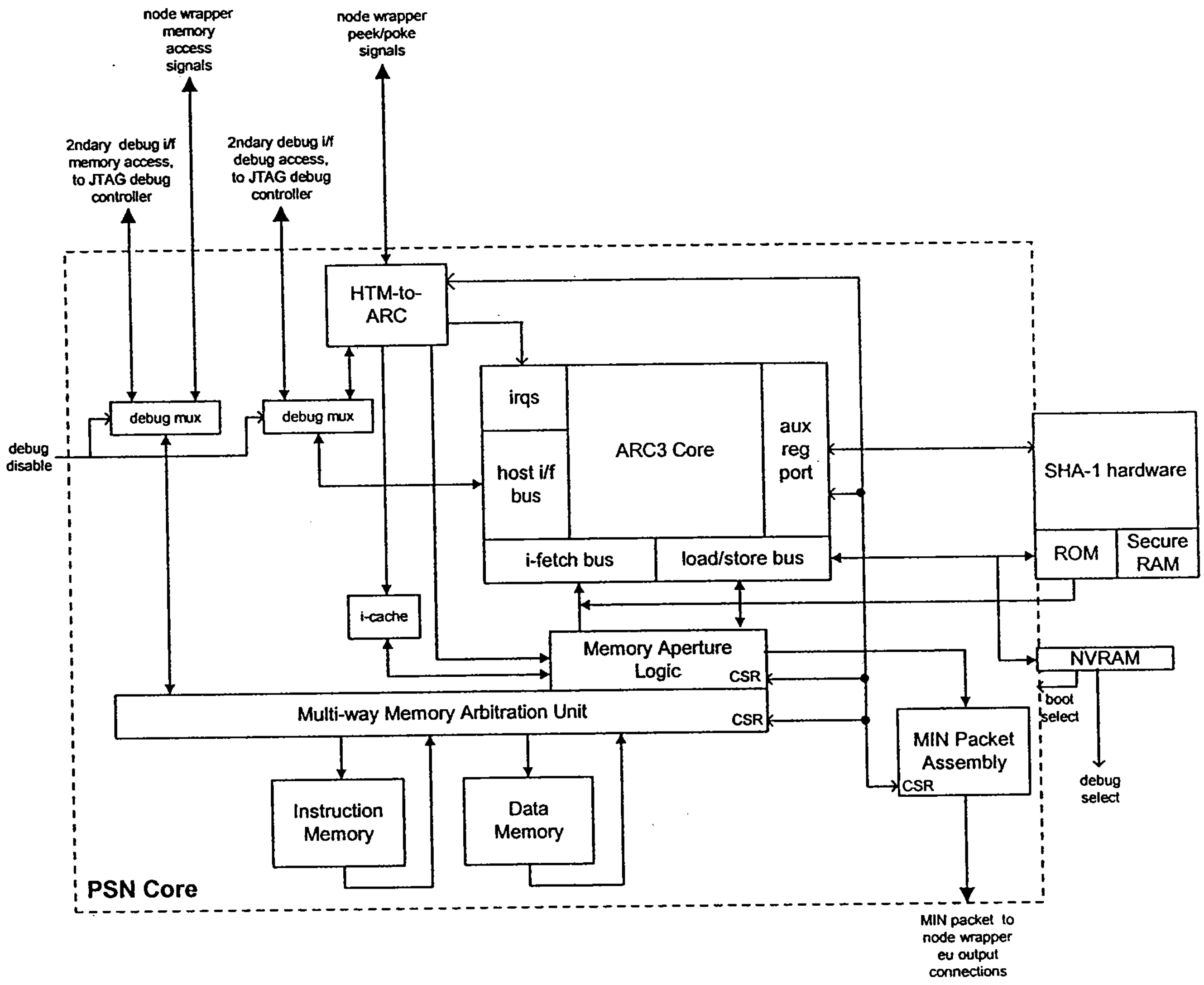
Or, when NODE_ENABLED signal from the node wrapper is cleared.

Furthermore, all memory chip enable (or clock enable) signals are deactivated to power down the memories whenever not in used.

2. K-node Specific Features

This section describes K-node specific features that are not shared with the PSN or that are different from the PSN described in the previous section. The unique elements different from the PSN are mainly the security elements. Since K-node and PSN use the same PSN Core and the same Node Wrapper to connect to the MIN, the determine factor for a PSN to be a K-node is the security bit in the Security Configuration Register shown in Table 1. This bit would determine whether the MIN words sent out by the MIN Packet Assembly block has the security bit set and whether it can produce the full range of MIN service words.

Figure 23: K-node Block Diagram

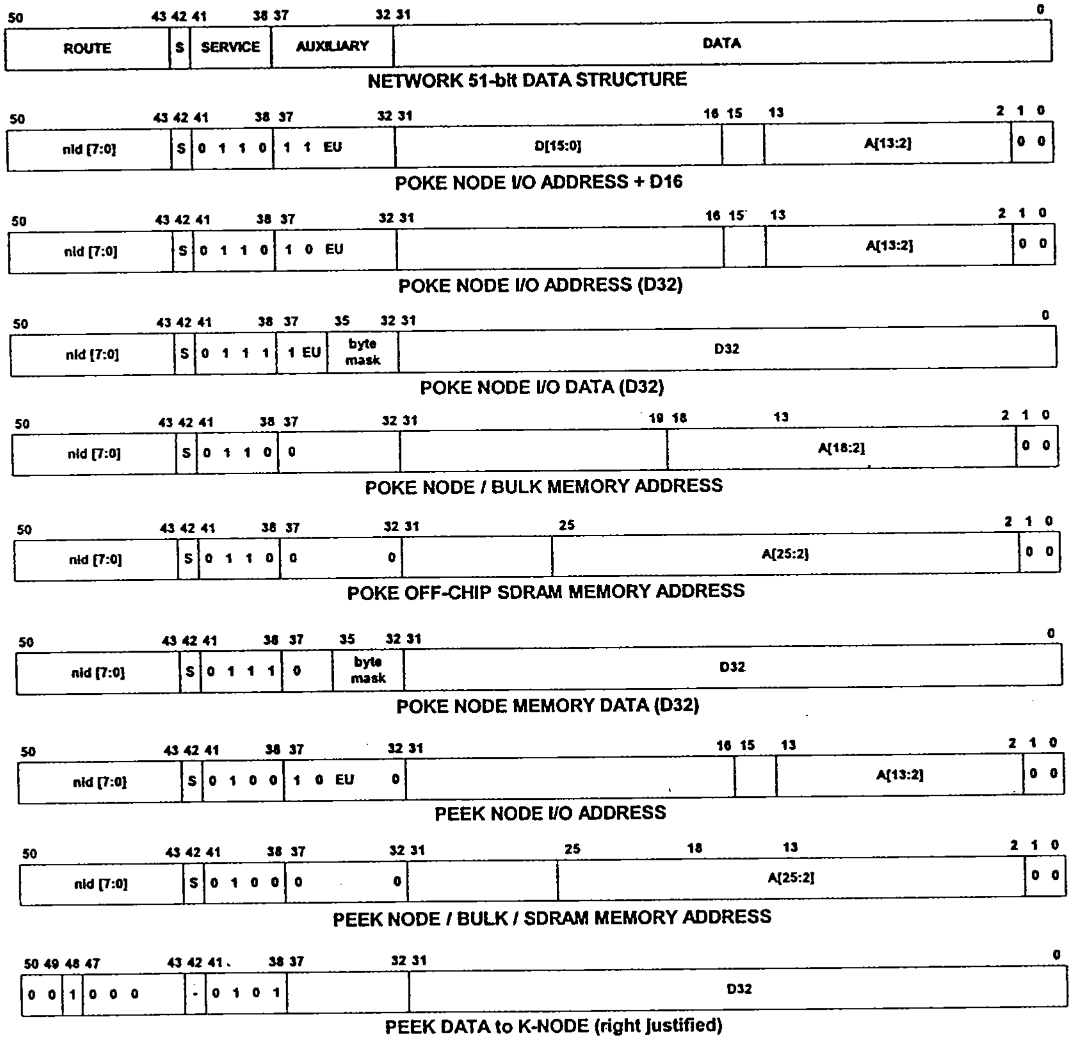

node wrapper memory access signals
node wrapper peek/poke signals
2ndary debug i/f memory access, to JTAG debug controller
2ndary debug i/f debug access, to JTAG debug controller
HTM-to-ARC
debug mux
debug mux
debug disable
irqs
ARC3 Core
aux reg port
host i/f bus
i-fetch bus
load/store bus
SHA-1 hardware
ROM
Secure RAM
i-cache
Memory Aperture Logic
CSR
NVRAM
boot select
debug select
Multi-way Memory Arbitration Unit
CSR
MIN Packet Assembly
CSR
Instruction Memory
Data Memory
PSN Core
MIN packet to node wrapper eu output connections

2.1 Functional Description

2.1.1 Memory System

In additional to bulk and external memory random accesses in PSN, the K-node posses the ability to peek and poke Node I/O and Node Memory spaces. However, all memory mapped accesses regardless of peek/poke or memory random access is done through ARC load and store instructions. The address appeared on the load/store bus is used to decoded what kind of MIN service to use for a given load or store request. If it is accessing the actual bulk or external memories, the memory random access MIN service will be used. However, for peeking and poking into the Node I/O space and node memories, the following MIN format will be send (Figure 24).

Figure 24: K-node Peek/Poke Formats on the MIN



NETWORK 51-bit DATA STRUCTURE

POKE NODE I/O ADDRESS + D16

POKE NODE I/O ADDRESS (D32)

POKE NODE I/O DATA (D32)

POKE NODE / BULK MEMORY ADDRESS

POKE OFF-CHIP SDRAM MEMORY ADDRESS

POKE NODE MEMORY DATA (D32)

PEEK NODE I/O ADDRESS

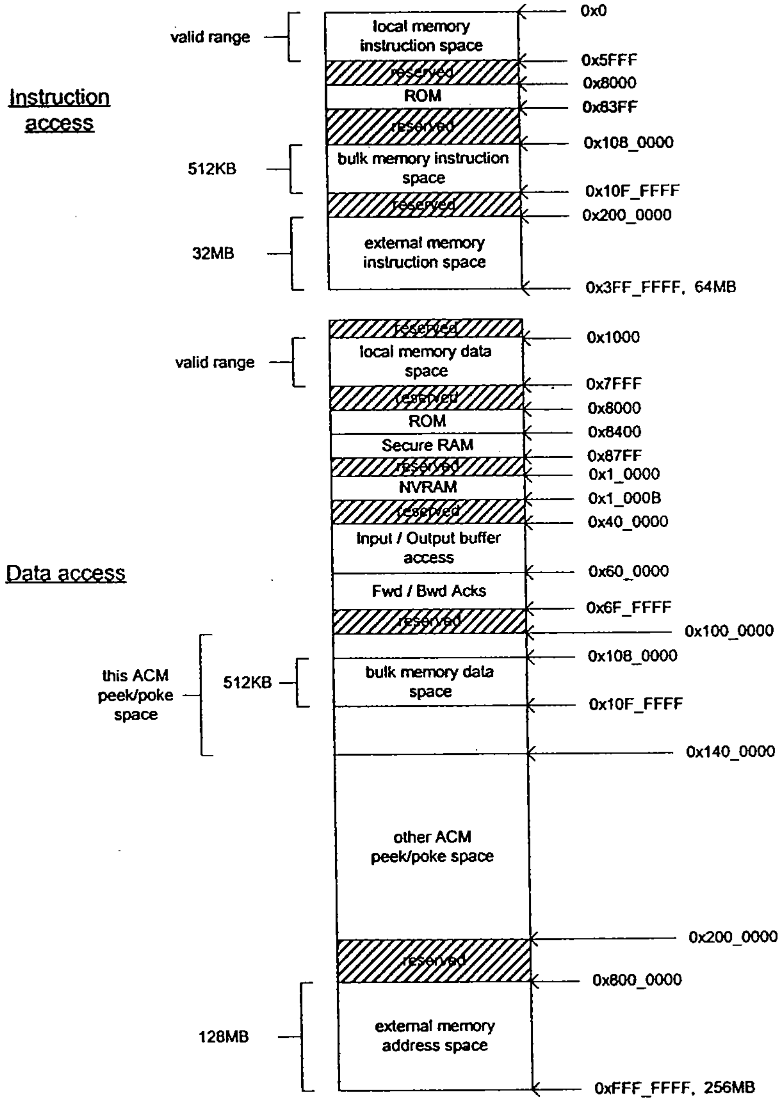
PEEK NODE / BULK / SDRAM MEMORY ADDRESS

PEEK DATA to K-NODE (right justified)

In the case of poke to the Node I/O space, one can use either STW or ST instructions. Depending on which instruction is used, a "POKE NODE I/O ADDRESS +D16" or "POKE NODE I/O ADDRESS (D32)" will be sent. If a "POKE NODE I/O ADDRESS +D16" format is sent, only the least significant 16 bits of the data is sent out onto the network. For some unknown reason that a STB instruction is used to poke, the byte data will be zero-extended and "POKE NODE I/O ADDRESS +D16" format is used.

In the case of poke to the Node Memory space, the poke will always be done on the dword address boundary using the "POKE NODE MEMORY ADDRESS" format. One should always use ST instruction to write the whole dword to the dword memory address for network bandwidth efficiency. In the event that a STW or STB instruction is needed and used, the valid data is located at the proper portion of the 32-bit word indicated by the byte mask bits. For peeks to Node I/O space, any size data returned is assumed to be right justified.

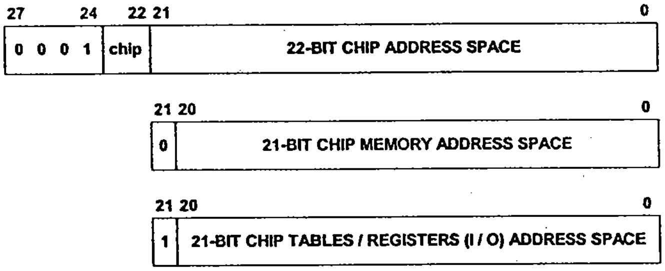
Figure 25: Default K-node Memory Map



In ACM chip peek/poke address space (0x100_0000 to 0x13F_FFFF for chip 0), the node memories, bulk memory, tables, and registers are mapped into this 22-bit contiguous chunk of address space in the K-node, taking into account of possible 2-tile memory aggregation mode. Therefore, on the load/store bus, the address will appears as shown in Figure 26, where bits

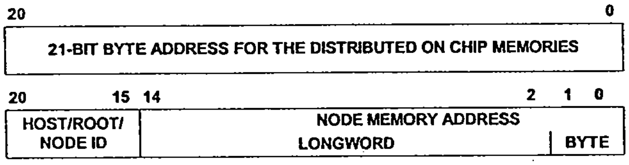
[23:22] identifies which ACM chip, and bit [21] indicates peeks and pokes are for node memory address space or the node i/o address space for registers and tables.

Figure 26: Memory Map for Chip Memory and Node I/O Space


27
24
22 21
0
0 0 0 1
chip
22-BIT CHIP ADDRESS SPACE
21 20
0
0
21-BIT CHIP MEMORY ADDRESS SPACE
21 20
0
1
21-BIT CHIP TABLES / REGISTERS (I / O) ADDRESS SPACE

First, the 21-bit chip memory address space is used by K-node to access the distributed On Chip Memories. This address closely correlates with the tile / memory block numbering convention shown in the "ACM Nodal Architecture: The Node Wrapper" document. The mapping of tile numbers and memory block numbers to the 21-bit address space is shown in Figure 27.

Figure 27: Address Map for Distributed On-Chip Memories



The Node I/O address space where all the table and registers reside is allocated to the upper portion of this peek/poke address space; for example, for ACM chip 0 it will be from 0x120_0000 to 0x13F_FFFF. The address break down is shown in Figure 28. The NODE field represents the 6-bit routing information for the MIN. Some special access targets need to be pointed out; they will have the following bits in the NODE field:

1. On chip Bulk Memory Controller I/O register access
   NODE field: 1 0 1 1 x x
2. External Memory Controller I/O register access
   NODE field: 1 0 1 1 x x
3. Host access
   NODE field: 1 1 x x x x
4. Permission Register in the Network's System_out module
   NODE field: 1 1 1 1 0 0

The least significant 14-bit address for these special (or non-nodal) access targets remains TBD by the target architecture. For example, how the External Memory Controller maps a peek/poke address to its internal control and status register is up to the Memory Controller designer. The only exception is peeking or poking to the Permission Register at the System_out module, where the least significant 14 address bits are unused.

Figure 28: Memory Map for Node I/O Space

21	20	14	13 7	6 2	1 0	
1	20-BIT CHIP TABLES / REGISTERS (I / O) ADDRESS SPACE					
1	ROUTE[5:0]	0	0 0 0 0 0 1 0	0 0 0 0 0	0 0	NCSR
1	ROUTE[5:0]	0	0 0 0 0 0 1 0	0 0 0 0 1	0 0	TPTBR
1	ROUTE[5:0]	0	0 0 0 0 0 1 0	0 0 0 1 0	0 0	TAR
1	ROUTE[5:0]	0	0 0 0 0 0 1 0	0 0 0 1 1	0 0	RRQR
1	ROUTE[5:0]	0	0 0 0 0 0 1 0	0 0 1 0 0	0 0	DCSR
1	ROUTE[5:0]	0	0 0 0 0 0 1 0	0 0 1 0 1	0 0	DSAR
1	ROUTE[5:0]	0	0 0 0 0 0 1 0	0 0 1 1 0	0 0	DASR
1	ROUTE[5:0]	0	0 0 0 0 0 1 0	0 0 1 1 1	0 0	DTCR
1	ROUTE[5:0]	0	0 0 0 0 1 0 0	CCT	0 0	
1	ROUTE[5:0]	0	0 0 0 0 1 0 1	PCT	0 0	
1	ROUTE[5:0]	0	0 0 0 1 0 0 0	STATE	0 0	
1	ROUTE[5:0]	0	0 0 1 0 0 0 0	RRQ	0 0	
1	ROUTE[5:0]	0	0 1 0 0 0 0 0	PTT-ADDR	0 0	
1	ROUTE[5:0]	0	0 1 0 0 0 0 1	PPT-SIZE	0 0	
1	ROUTE[5:0]	1	NODE-SPECIFIC REGISTERS			

2.1.1.1 Security Memories

The extra features in the K-node, the embedded ROM and Secure RAM, two consecutive memory regions respectively, are connected to ARC local load/store bus, which bypasses all memory arbitration and are accessed directly. For load/store access, the base address of these two regions can be relocated through writing to auxiliary register 0x18, in 2kB (0x800) address boundaries. The default base address for these two memory would be 0x8000, thus for ROM access: 0x8000 ~ 0x83FF, and for secure RAM access: 0x8400 ~ 0x87FF. If the content of auxiliary register 0x18 is changed, the base memory address for these two memories will change in tandem.

However, both instruction and load/store access can read the content of the ROM through the same address range. For ifetch, the ROM base address is fixed at the default 0x8000. The TOP 192 bytes of the ROM are occupied by the pico-codes needed to run the SHA-1 hardware. The TOP 256 bytes of Secure RAM are shared with SHA-1 hardware since it uses this region for processing message digest; both ARC and the SHA-1 have access to these regions of memory. However, neither the MIN nor the secondary debug port can peek or poke directly to these two regions.

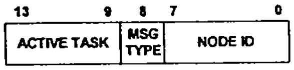
2.1.2 Messaging System

Table 30: Messaging Auxiliary Registers

Number	Name	Function
0x124	MSG_RD	Read only, pointer to queued receive message FIFO buffer on the K-node, 14-bit

When a message request is received at the K-node, the request is queued in a FIFO where MSG_RD auxiliary register serves as the read port for this circular buffer. The FIFO is preliminary set at 14x32 deep allowing up to maximum of 32 message requests from different nodes to queue up. The buffer content is 14-bit wide (5-bit Active Task ID, 1-bit Message Type, 8-bit source Node ID, Figure 29) that identifies which node was sending a message and the message code. Whenever the FIFO is not empty, meaning there are valid messages in the queue, interrupt 5 will remain asserted and de-assert when the FIFO is again empty. In the case when message queue overflows (the FIFO is full), an interrupt (#7) will also be generated. This interrupt will only be de-asserted when the FIFO read pointer is advanced enough so that the FIFO is not full. Every read from the MSG_RD auxiliary register advances the FIFO's read pointer by one.

Figure 29: Message Receive FIFO Data Format



The message type bit are defined as follows:

Message Type	Description
0	eu_message
1	dma_done_message

For the PSN, its specific message codes can reside in the 3-bit MSG_MB node specific register for the K-node to retrieve as mentioned previously. An example of what the 3-bit message code (agreed upon kernels running on the K-node and PSN) can represent is as follows:

Message	Description
0x0	---
0x1	memory exception
0x2	instruction exception
0x3	timer/counter 0 wrap around
0x4	timer/counter 1 wrap around
0x5 - 0x7	---

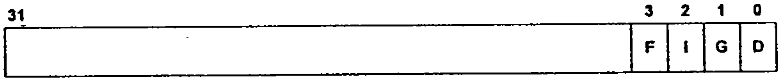
2.1.3 Security Hardware

The SHA-1 processor for secure hash will be control through auxiliary register access.

Table 31: SHA-1 Control Register in Auxiliary Space

Number	Name	Function
0x80	SHA_CSR	Control and status register for SHA-1 hardware

Figure 30: SHA-1 Control Register Format



D = Done bit => Read only, 0 = working, 1 = done.
G = Go bit => Write only, write 1 to start SHA-1 controller
I = Init_it bit => Write only, write 1 to initialize SHA-1 controller
F = Frame number to process => Write only

Again, SHA-1 processor uses the TOP 256 bytes of Secure RAM for processing message digest. Both ARC and the SHA-1 have access to this region of memory. One could deposit the next block of data to be processed at the unused frame location while another block of data is being processed. The ARC local load/store access will always be stalled if the SHA-1 processor is accessing the Secure RAM. If the Secure RAM remains at the default location, its usage map by the SHA-1 processor is shown below:

Table 32: SHA-1 Secure RAM Usage Map

Address	Content	Address	Content	Address	Content
0x8700	W_0	0x8740	a	0x8780	W_0
0x8704	W_1	0x8744	b	0x8784	W_1
0x8708	W_2	0x8748	c	0x8788	W_2
0x870c	W_3	0x874c	d	0x878c	W_3
0x8710	W_4	0x8750	e	0x8790	W_4
0x8714	W_5	0x8754		0x8794	W_5
0x8718	W_6	0x8758	K_0	0x8798	W_6
0x871c	W_7	0x875c	K_1	0x879c	W_7
0x8720	W_8	0x8760	H_0	0x87a0	W_8
0x8724	W_9	0x8764	H_1	0x87a4	W_9
0x8728	W_{10}	0x8768	H_2	0x87a8	W_{10}
0x872c	W_{11}	0x876c	H_3	0x87ac	W_{11}
0x8730	W_{12}	0x8770	H_4	0x87b0	W_{12}
0x8734	W_{13}	0x8774		0x87b4	W_{13}
0x8738	W_{14}	0x8778	K_2	0x87b8	W_{14}
0x873c	W_{15}	0x877c	K_3	0x87bc	W_{15}

Example procedures for performing SHA-1 secure hash using this processor:

1. Initialize H_0 through H_4

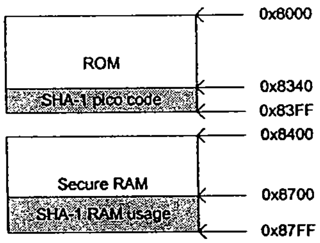
```
mov     %r0, 0x8760
st      0x67452301,  [%r0]
st.a    0xefcdab89,  [%r0,4]
st.a    0x98badcfe,  [%r0,4]
st.a    0x10325476,  [%r0,4]
st      0xc3d2e1f0,  [%r0,4]
```

2. Deposit the first 512-bit message block to Frame 0 message data space: 0x8700 ~ 0x873C, using LD and ST instructions
3. Start SHA-1 with initialization; note processing Frame 0 first
   ```
   sr      6, [SHA_CSR]
   ```
4. While SHA-1 processor is running, deposit the second 512-bit message block to Frame 1 message data space: 0x8780 ~ 0x87BC, using LD and ST instructions
5. When finished loading the second message block, poll for SHA-1 done indication
   ```
   poll_done:    lr     %r0, [SHA_CSR]
                 and    0, %r0, 0
                 beq     poll_done
   ```
6. If SHA-1 processor is done with processing the first block, start SHA-1 with Frame 1 processing but no initialization
   ```
   sr      10, [SHA_CSR]
   ```
7. Repeat the loading and processing of the 512-bit message block, alternating between Frame 0 and Frame 1

The final 160-bit message digest output resides in dword location a through e (address 0x8740 ~ 0x8750). The pico code program that runs the SHA-1 processor should reside on the TOP 192 bytes of the ROM, in the address range 0x8340 ~ 0x83FF. For details of how to program the SHA-1 processor please consult the "Hardware Implementation of the Secure Hash Standard" document.

To summarize, the free ROM and Secure RAM range that is available to the OS developer, excluding the SHA-1 occupied space, is shown in Figure 31. Note, programs running on the ARC still can read from all ROM and Secure RAM addresses and write to all Secure RAM addresses.

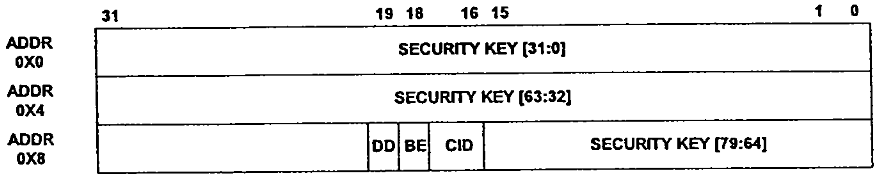
Figure 31: ROM and Secure RAM Usage Map



2.1.3.1 Nonvolatile Memory

The security planning for the ACM expects an internal nonvolatile memory (NVRAM) to be present in the chip. This section discusses the usage of such a memory. Similar to the Secure RAM, the NVRAM will be connected to ARC local load/store bus, which bypasses all memory arbitration and are accessed directly. Only a program running on the K-node can access the content of the NVRAM; and neither the MIN nor the secondary debug port can access directly into this memory. The physical size of NVRAM proposed is 84 x 1, in which the security key takes the least significant 80 bits (Figure 32). However, the K-node can access it using address range is from 0x10000 to 0x1000B.

Figure 32: NVRAM Address Map



DD = Debug Disable (JTAG ON / OFF)
BE = Boot Enable (Halt / Run at ROM) on reset
CID = Chip ID for comparison

The top word in the NVRAM contains security parameters Debug Disable (DD), Boot Enable (BE), and Chip ID (CID). These parameters take on the following value by default:

	Debug Disable	Boot Enable	Chip ID	Comments
Default	JTAG (0)	Halt (0)	00	Assumes NVRAM default state = 0
	MIN	Boot on Reset	Other ID	

When Debug Disable is in the default state, the secondary JTAG debug access to K-node internal registers and memories is enabled. The JTAG debug and MIN accesses share the same connections into the ARC. Therefore, they are multiplexed, where a MIN access will always halt the JTAG access. When Debug Disable is asserted, the MIN access is the only gateway into K-node internal registers and memories. The secondary debug interface from the JTAG controller is meant for bring-up and initial debugging of codes on the K-node. For the full chip debug, one should use the MIN system connection for efficiency, by setting this Debug Disable bit.

In the PSN, the ARC will default to "halt" upon reset. However, in the K-node, this is controlled by the Boot Enable bit in the NVRAM. The default NVRAM content still makes the K-node to power up to "halt", and allow the debugger to start it by writing to ARC status register. When this Boot Enable bit is set, the K-node would boot from the hard-coded reset vector pointing to the internal ROM location (address 0x8000).

The 2-bit Chip ID in the NVRAM is used to verify if the current chip is chip 0. Only chip 0 has a valid K-node that can turn on the permission register on the system bus to allow host access into the ACM chip.

2.1.3.2 Boot Select Pin

The state on the Northstar chip BOOT_SEL pin will have a software window for the boot code running on the K-node. This is done through the following auxiliary register access.

Table 33: Boot Select Pin Auxiliary Register Access

Number	Name	Function
0x81	BOOT_SEL	Chip Boot_sel pin state, 1-bit

The initial boot loader residing on the internal ROM can read the state of this pin, and decide the location where the rest of the boot program resides, either present in the flash memory attached to the Northstar chip or housed in the host processor. For example, if the BOOT_SEL pin is tied low externally, this can indicate the presence of a flash memory connected to the external memory controller with the boot codes programmed in it. Thus, the initial boot loader can jump to the flash memory location and execute the rest of the boot sequence. On the other hand, if this pin is tied high externally, it indicates the absence of the flash memory connected to Northstar. Thus, the initial boot loader waits for the host processor to give the K-node the start location of the rest of the boot codes.

2.1.4 MIN Transactions

When the Security Configuration Register bit is set in the K-node, MPA can generate all MIN services targeting to all MIN space, including this ACM's all address and Node I/O range plus potentially other ACM's all address range. In addition, the MIN words sent out have the security bit set.

2.1.5 DMA Transfer

The Node Wrapper DMA Engine will be used for transferring from local memory to other nodes and memories on the MIN. The DMA Engine can be controlled by K-node poking into its own Node Wrapper, similar to how it starts other nodes' DMA engines. Note, this is allowed since K-node has its security bit is set.

2.2 Node Size Estimate

The following table estimates the PSN die size including the security components. Similarly, this estimate does not include the area required by the node wrapper elements or the node memory allocated. The embedded NVRAM is not included in the estimate, since its inclusion is TBD; its estimated area is included for reference only.

Table 34: Node Size Estimate in 0.13um

Reg FF	3.7	9.7	35.6	Artisan DFFRHQX1
Multiplexer, 4 inputs: per input	3.7	2.2	8.1	Artisan MX4X1
Adder (full, bit slice)	3.7	14.7	54.3	Artisan ADDFHX2
SRAM cell, 1 KB (256x32)	125	405	50737	Artisan SRAM-SP-HS
Regfile cell, 2-port, 32x32	108	133	14361	Artisan HS-RF-2P
Regfile cell, 1-port, 16x16	84	75	6310	Artisan HS-RF-1P
ROM cell, 256x32	130	280	36470	Artisan ROM-DIFF-HS
NVRAM, 10x17	409	430	175870	Virage NOVeA RAM
Design Elements				
Element	Size	Routing	Sq um	Comments
Gate, in layout @ 70% wiring	5.1	1.43	7.3	Artisan 2-input NAND
Multiplexer	8.1	1.43	11.5	Artisan multiplexer
Reg FF	35.6	1.60	57.0	Artisan DFF
Node Size Estimate				
Element	Gate	Size	Sq um	Comments
ARC 3 basecase core	8,400	7.3	61,106	using Gate
ARC standard extensions:				
32x32 barrel shifter	2,100	7.3	15,277	using Gate
32x32 multiplier	12,900	7.3	93,842	
swap	200	7.3	1,455	
min/max	500	7.3	3,637	
normalize	400	7.3	2,910	
I-Cache logic	1,600	7.3	11,639	using Gate
Memory Arbitration Unit	2,000	7.3	14,549	
Memory Aperture Logic	3,000	7.3	21,824	
Node wrapper adapter logic:				
MIN packet assembly	3,500	7.3	25,461	using Gate
HTM-to-ARC	3,000	7.3	21,824	
Security Hardware:				
SHA-1 accelerator	1,500	7.3	10,912	using Gate
control	1,000	7.3	7,275	
Total logic size	**40,100**		**291,710**	
Element	Size		Sq um	Comments
Register File	1,024		28,721	Artisan RF-2P
I-Cache data (1 KB)	8,192		50,737	Artisan SRAM-SP
I-Cache tag	240		6,310	Artisan RF-1P
secure memory (1 KB)	8,192		50,737	Artisan SRAM-SP
ROM (1 KB)	8,192		36,470	Artisan ROM
Total memory size			**172,975**	
Size @ 0.13, sq um			464,685	
Size @ 0.13, sq mm			0.465	
Layout Margin			10%	
Size sq mm @ 0.13um, sq mm			**0.511**	

Element	Gate	Size	Sq um	Comments
MIN interface (Node Wrapper)	32000	7.3	233,600	using COSM sMIN

3. Assumptions and Issues

This architectural proposal requires a few minor modifications to the standard Node Wrapper for the one attached to both the K-node and PSN, for the purpose of allowing K-node and PSNs to share the same and maximum number of adapter blocks:

1. Instead of the Security bit be hard-coded in the Output Pipeline Register block, this bit needs to be supplied as a connection (SECURE_BIT) from the PSN_CORE to the Node Wrapper Interface. Therefore, NODE_OUT_DATA[42] and OUT_PIPE_DATA[42] signals get their values from this register, as well as the internal loop back of the MIN word from the aggregator to the distributor in the Node Wrapper for pokes targeted at its own wrapper. This would allow the MIN Packet Assembly block to generate the necessary MIN word to its own node wrapper, e.g. modification of the task GO-bit by the SoftKernel.
2. Decoder in the Node Wrapper needs to provide Peek Return (PEEK_RETURN) signaling, and all auxiliary field (REG_IN_O[37:32]) needs to be available from the Node Wrapper Interface also. This is largely for K-node functionality.
3. Incoming Message (MSG_IN) signaling needs to be available from the Node Wrapper Interface. Thus, the K-node can be triggered to service a message sent by others. Again, the auxiliary field of the MIN word (REG_IN_O[37:32]) and the data field (REG_IN_O[31:0]) contain the incoming information for K-node to decode the message internally.